FinalProject

December 15, 2024

[1]:
```
# All necessary packages needed
import pandas as pd
import numpy as np
import matplotlib
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split, KFold
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, MinMaxScaler,␣
 ↪StandardScaler
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪ConfusionMatrixDisplay
from sklearn.model_selection import ParameterGrid
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
import warnings
from sklearn.exceptions import ConvergenceWarning
import shap
```

[2]:
```
# import data
df = pd.read_csv('data/kidney_disease.csv')
df.head()
# EDA - print rows and columns
print("Rows:", df.shape[0], "Columns:", df.shape[1])
```

```
Rows: 400 Columns: 26
```

0.1 Exporatory Data Analysis (EDA)

0.1.1 (i) About the whole data structure

[3]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
```

```
Data columns (total 26 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id              400 non-null    int64
 1   age             391 non-null    float64
 2   bp              388 non-null    float64
 3   sg              353 non-null    float64
 4   al              354 non-null    float64
 5   su              351 non-null    float64
 6   rbc             248 non-null    object
 7   pc              335 non-null    object
 8   pcc             396 non-null    object
 9   ba              396 non-null    object
 10  bgr             356 non-null    float64
 11  bu              381 non-null    float64
 12  sc              383 non-null    float64
 13  sod             313 non-null    float64
 14  pot             312 non-null    float64
 15  hemo            348 non-null    float64
 16  pcv             330 non-null    object
 17  wc              295 non-null    object
 18  rc              270 non-null    object
 19  htn             398 non-null    object
 20  dm              398 non-null    object
 21  cad             398 non-null    object
 22  appet           399 non-null    object
 23  pe              399 non-null    object
 24  ane             399 non-null    object
 25  classification  400 non-null    object
dtypes: float64(11), int64(1), object(14)
memory usage: 81.4+ KB
None
```

[4]:
```
# EDA - target variable type
print("The column Y is discrete")
print(df['classification'].value_counts())
```

```
The column Y is discrete
classification
ckd       248
notckd    150
ckd\t       2
Name: count, dtype: int64
```

[5]:
```
df['classification'] = df['classification'].replace(to_replace={'ckd\t':'ckd',␣
  ↪'notckd':'not ckd'})
print(df['classification'].value_counts())
```

```
classification
```

```
ckd        250
not ckd    150
Name: count, dtype: int64
```

Hence we don't need stratified spliting. We will use basic split for train, val and test sets.

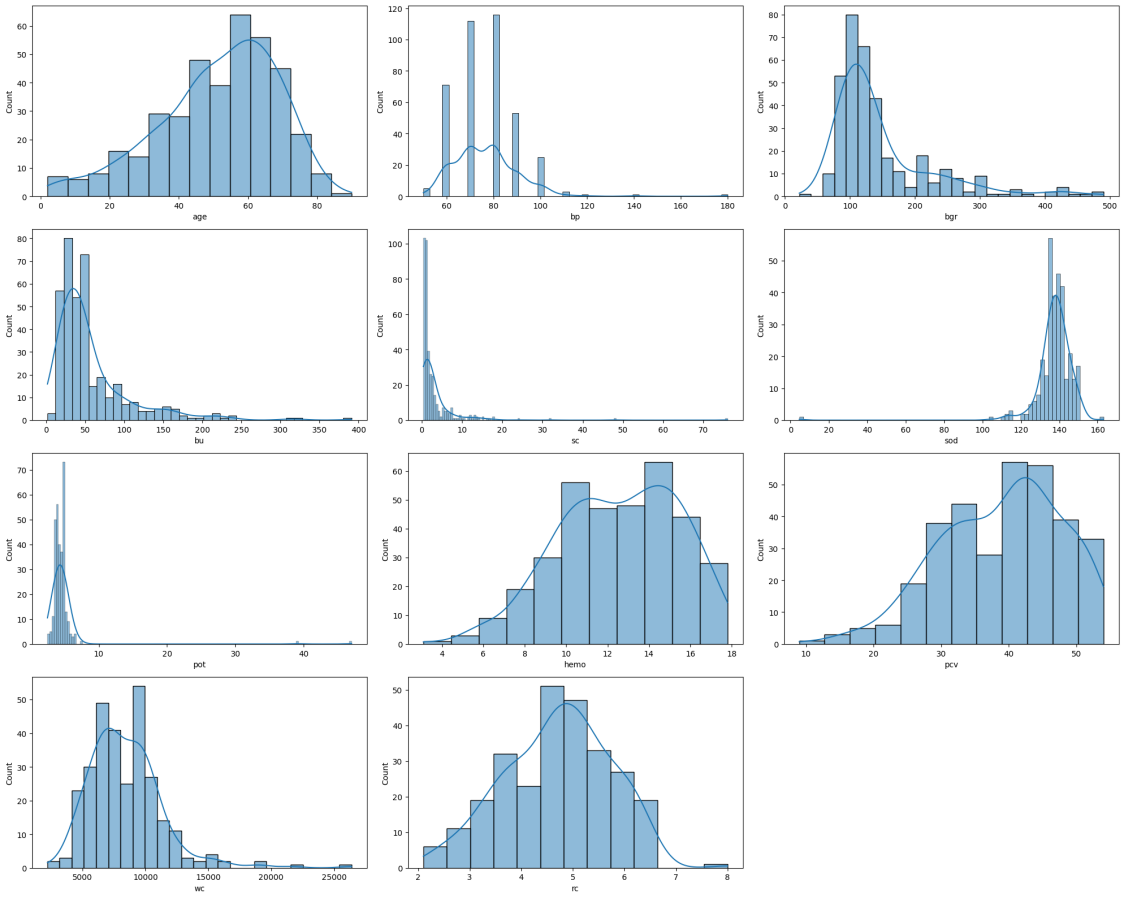
[6]:
```
# seperate category columns and numerical columns
num_cols = ['age', 'bp', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wc',␣
 ↪'rc']
cat_cols = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'pe', 'ane', 'appet',␣
 ↪'sg', 'al', 'su']

# check unique values in categorical columns
for col in cat_cols:
    print(f"{col} has {df[col].unique()}")
```

```
rbc has [nan 'normal' 'abnormal']
pc has ['normal' 'abnormal' nan]
pcc has ['notpresent' 'present' nan]
ba has ['notpresent' 'present' nan]
htn has ['yes' 'no' nan]
dm has ['yes' 'no' ' yes' '\tno' '\tyes' nan]
cad has ['no' 'yes' '\tno' nan]
pe has ['no' 'yes' nan]
ane has ['no' 'yes' nan]
appet has ['good' 'poor' nan]
sg has [1.02  1.01  1.005 1.015  nan 1.025]
al has [ 1.  4.  2.  3.  0. nan  5.]
su has [ 0.  3.  4.  1. nan  2.  5.]
```

0.1.2 (ii) Then we first take a look at the continuous variables

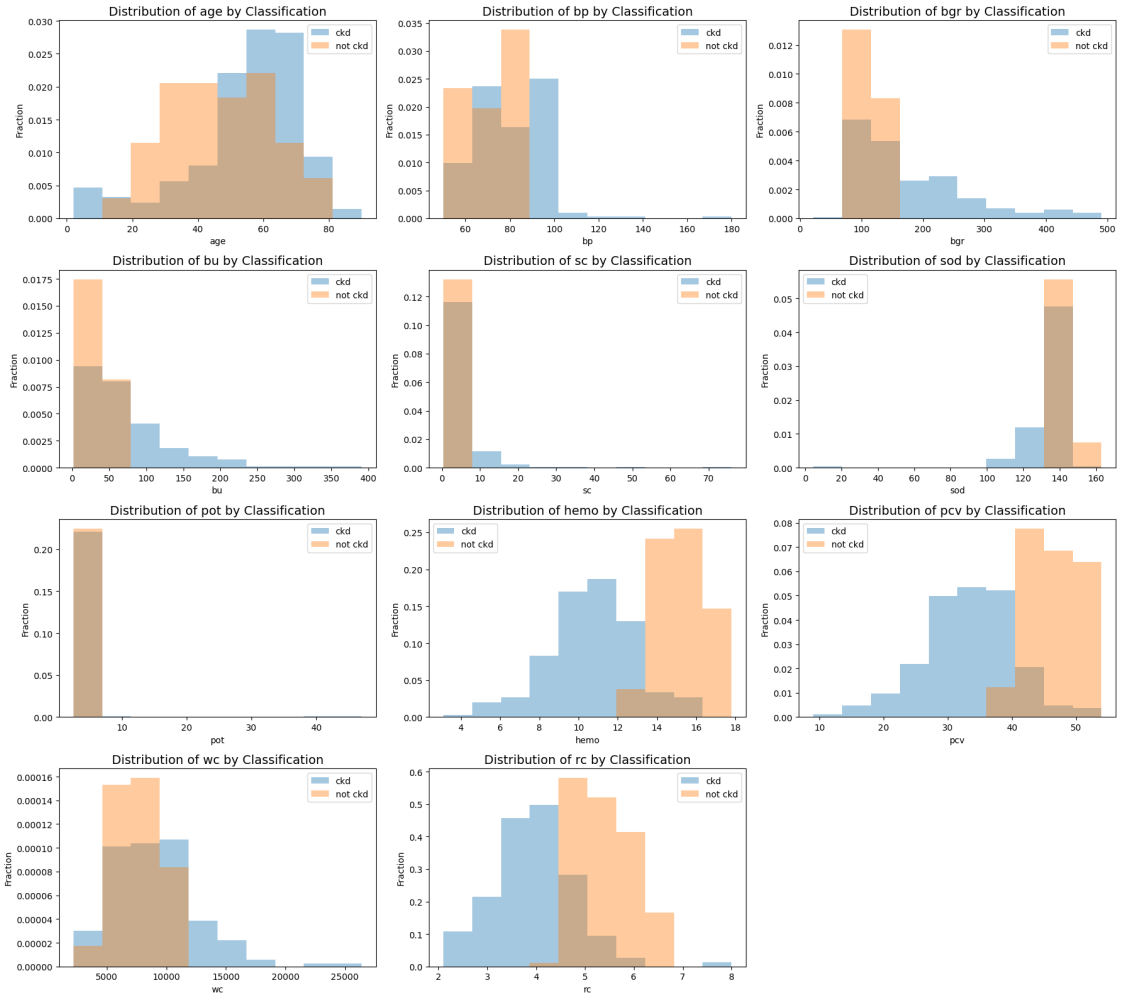
[7]:
```
# for the column pcv, wc, rc, there are some strings ? and some values with tab␣
 ↪ahead
for column in ['pcv', 'wc', 'rc']:
    df[column] = df[column].str.strip()  # Remove any leading/trailing␣
 ↪whitespace
    df[column] = df[column].replace('?',np.nan)

plt.figure(figsize=(20, 16))
plotnumber = 1

for column in num_cols:
    # Convert to numeric
    df[column] = pd.to_numeric(df[column])

    ax = plt.subplot(4, 3, plotnumber)
```

```
    sns.histplot(df[column].dropna(), kde=True)  # Use histplot instead of␣
 ↪distplot
    plt.xlabel(column)

    plotnumber += 1

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/cont_hist2.
 ↪png', dpi = 350)
plt.show()
```



[8]:
```
# compare the classification of continous variables
plt.figure(figsize=(18, 16))
categories = df['classification'].unique()

for i, column in enumerate(num_cols):
    plt.subplot(4,3,i+1)  # Create subplots
    bin_range = (df[column].min(), df[column].max())  # Set the bin range for␣
 ↪the current column
```

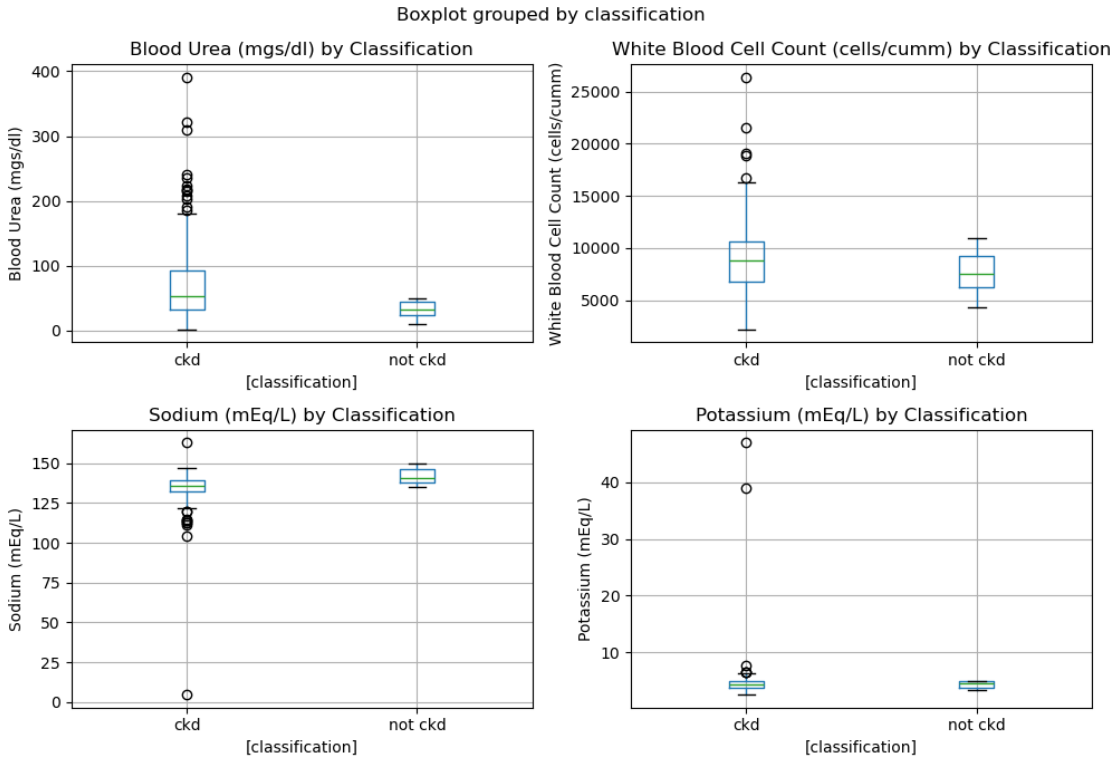
```
    for c in categories:
        plt.hist(df[df['classification'] == c][column], alpha=0.4, label=c,␣
 ↪range = bin_range, density=True)
        plt.legend()
        plt.ylabel('Fraction')
        plt.xlabel(column)
        plt.title(f'Distribution of {column} by Classification', fontsize = 14)

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/cont_hist.png',␣
 ↪dpi = 350)
plt.show()
```



Now we find that age, bp, bgr, hemo, pcv, wc, rc may have large influence on classification(has disease or not). And we are interested in using bar plot to see if any columns have too many

extreme outliers. From above plot, I will choose bgr, bu, wc to investigate.

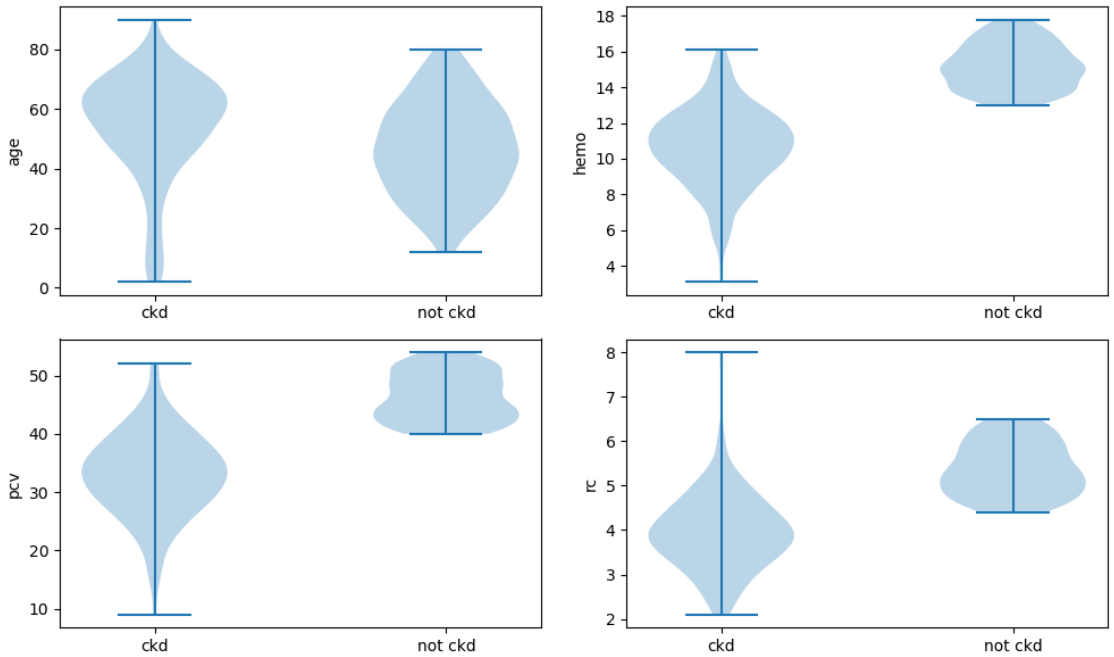
[9]:
```
plt.figure(figsize = (10,7))
bar_cols = ['bu', 'wc', 'sod', 'pot']
bar_cols_full = ['Blood Urea (mgs/dl)', 'White Blood Cell Count (cells/cumm)',␣
 ↪'Sodium (mEq/L)', 'Potassium (mEq/L)']
for i, col in enumerate(bar_cols):
    plt.subplot(2,2,i+1)
    df[[col, 'classification']].boxplot(by = 'classification', ax = plt.gca())
    plt.title(f'{bar_cols_full[i]} by Classification')
    plt.ylabel(bar_cols_full[i])

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/boxplot.png',␣
 ↪dpi = 350)
plt.show()
```



Next, I am going to visualize some distribution of some attributes across different classidication by violin plot. Here I choose age, hemo, pcv and rc since they have large gap between two distributions from

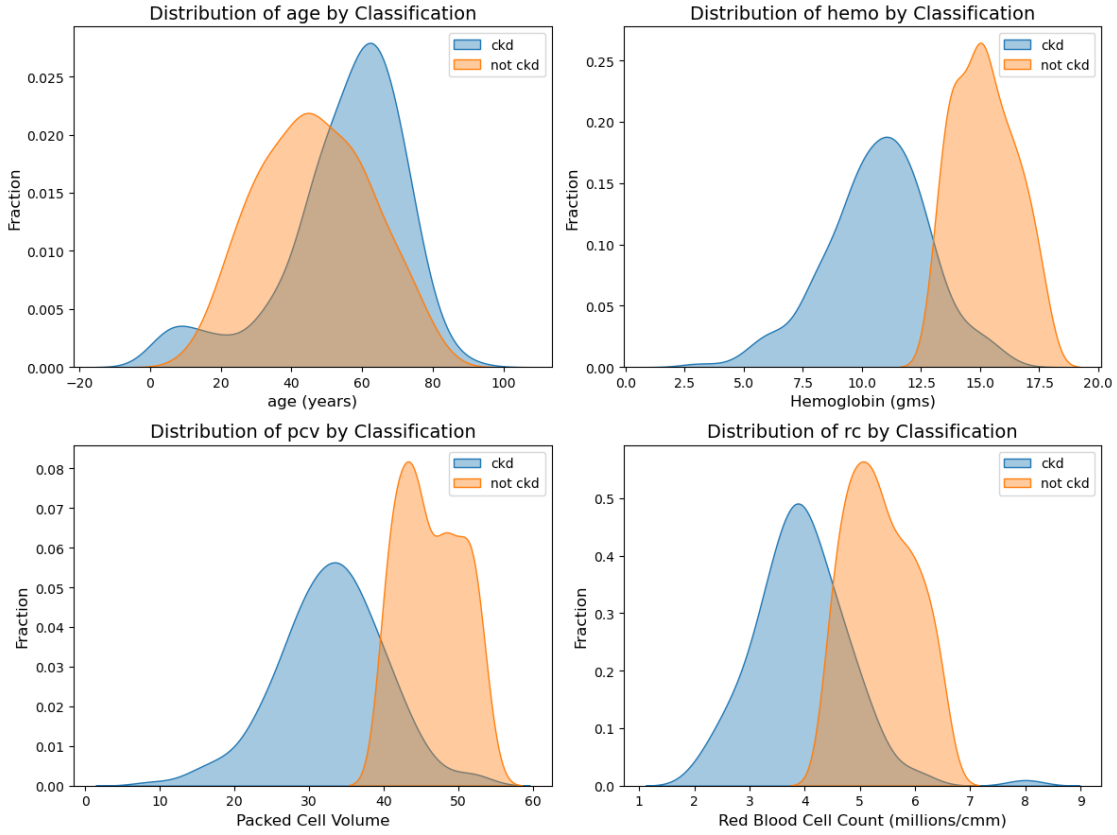
[10]:
```
# violin plot
plt.figure(figsize = (10,6))
```

```
vio_cols = ['age', 'hemo', 'pcv', 'rc']

for i, col in enumerate(vio_cols):
    plt.subplot(2,2,i+1)
    plt.violinplot(dataset = [df[col][df['classification'] == 'ckd'].dropna().
 ↪values,
                              df[col][df['classification'] == 'not ckd'].
 ↪dropna().values]) # dropna to deal with missing values
    plt.xticks([1,2], ['ckd', 'not ckd'])
    plt.ylabel(col)

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/violin.png',␣
 ↪dpi = 350)
plt.show()
```



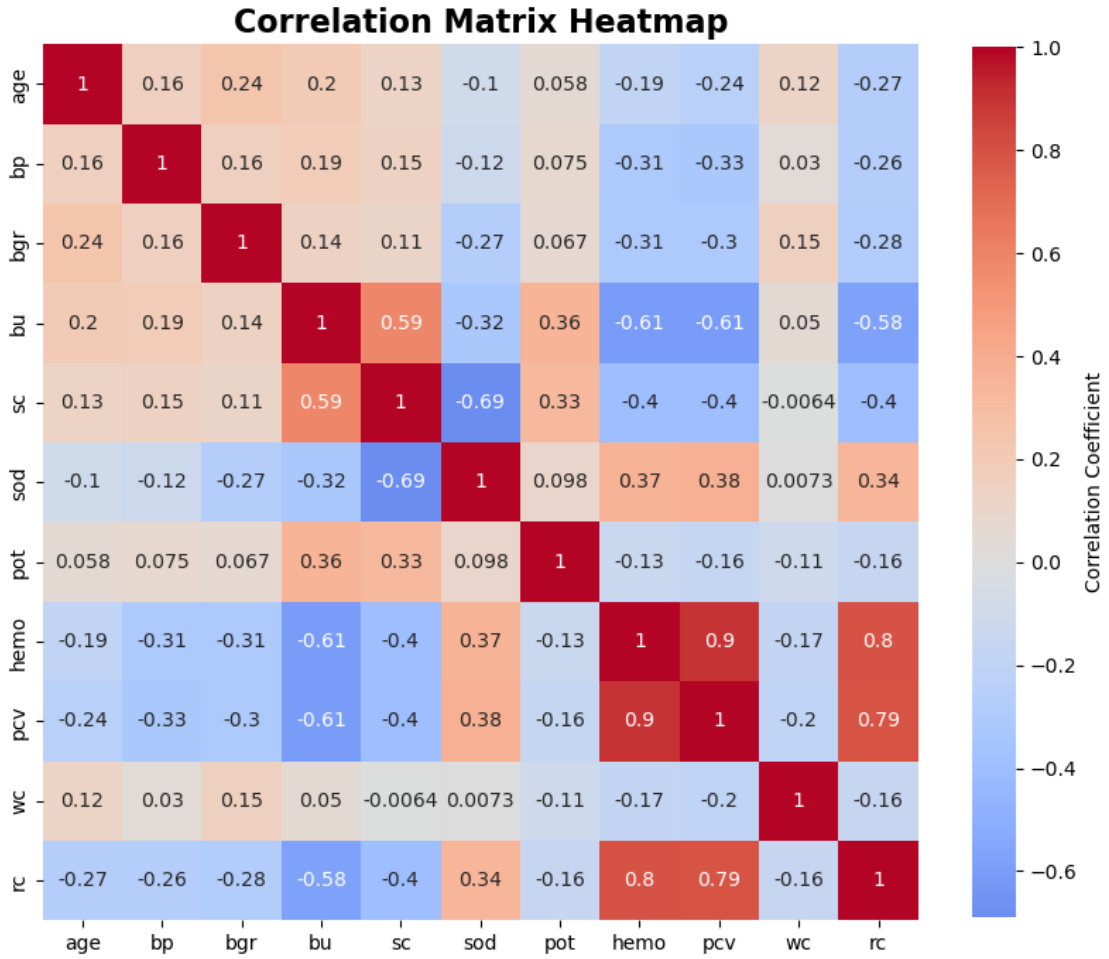
[11]:
```
# plot the same attributes using histogram
plt.figure(figsize=(12, 9))
categories = df['classification'].unique()
full_name_cols = ['age (years)', 'Hemoglobin (gms)', 'Packed Cell Volume', 'Red␣
 ↪Blood Cell Count (millions/cmm)']
for i, column in enumerate(vio_cols):
    plt.subplot(2,2,i+1) # Create subplots

    for c in categories:
```

```
        sns.kdeplot(df[df['classification'] == c][column], alpha=0.4, label=c,␣
 ↪fill=True)
        plt.legend()
        plt.ylabel('Fraction', fontsize = 12)
        plt.xlabel(full_name_cols[i], fontsize = 12)
        plt.title(f'Distribution of {column} by Classification', fontsize = 14)

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/cont_hist_small.
 ↪png', dpi = 350)
plt.show()
```



[12]:
```
df_corr_nums = df[num_cols].corr(method='pearson')
plt.figure(figsize=(10, 10))
sns.heatmap(df_corr_nums, annot = True, cmap = 'coolwarm', center=0,␣
 ↪cbar_kws={'label':'Correlation Coefficient', 'shrink': 0.8},
            square=True,  linecolor='black')

plt.title('Correlation Matrix Heatmap', fontsize=16, fontweight='bold')
```

```
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/heatmap.png',␣
  ↪dpi = 350)
plt.show()
```



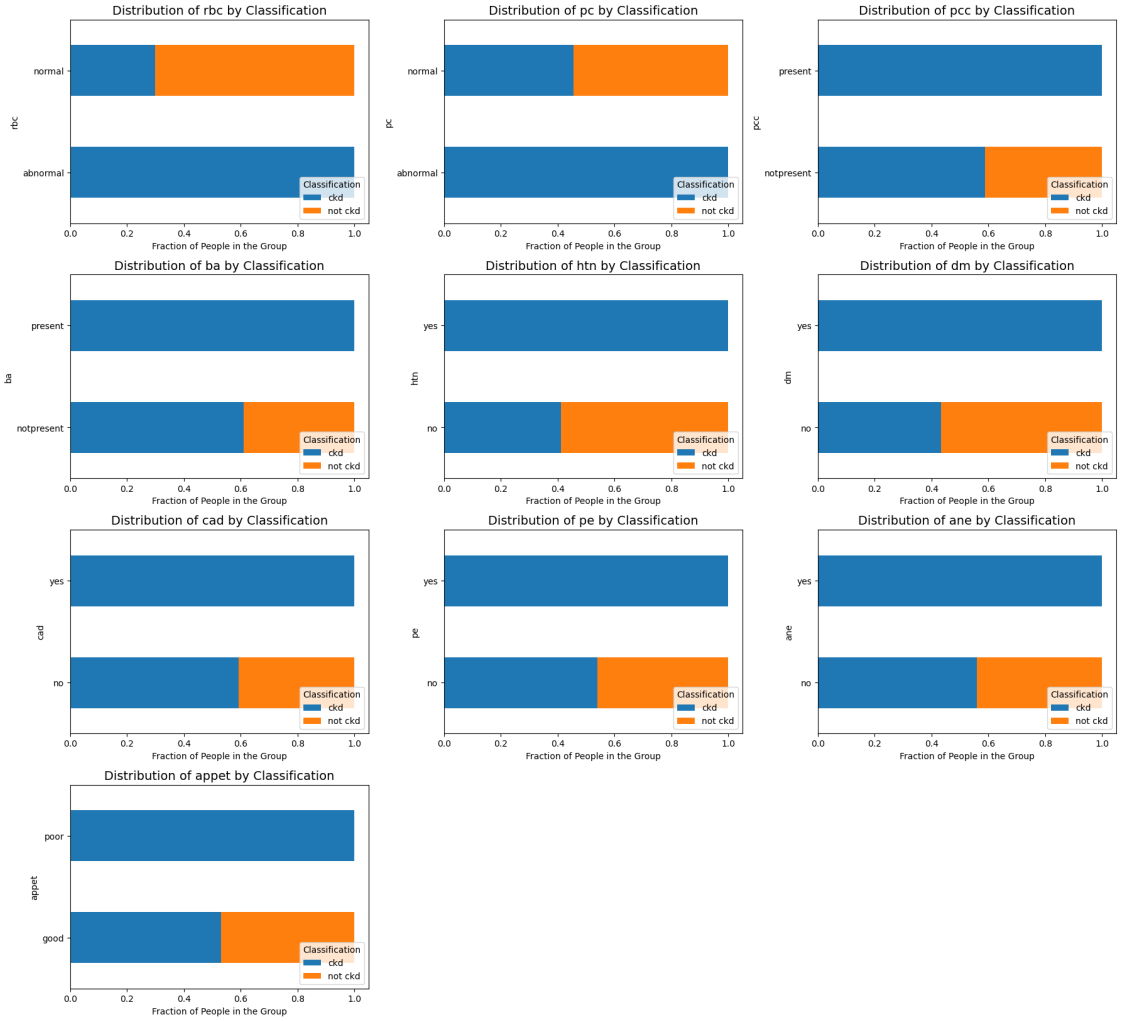
we find hemo and pcv have strong correlation relationship, hence we choose to drop one of these attributes – hemo.

[13]:
```
df = df.drop(columns = 'hemo')
num_cols.remove('hemo')
```

0.1.3 (iii) Next we take a look at the categorical variables with no order

[14]:
```
plt.figure(figsize=(18, 20))
categories = df['classification'].unique()
cat_cols = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'pe', 'ane', 'appet']

for column in cat_cols:
```

```
    df[column] = df[column].str.strip()  # Remove any leading/trailing␣
 ↪whitespace

for i, column in enumerate(cat_cols):
    plt.subplot(5, 3, i + 1)  # Use dynamic column count

    count_matrix = df.groupby([df[column], df['classification']]).size().
 ↪unstack()
    count_matrix_norm = count_matrix.div(count_matrix.sum(axis=1), axis=0)

    count_matrix_norm.plot(kind='barh', stacked=True, ax = plt.gca())  # Using␣
 ↪the current axis

    plt.legend(title='Classification', fontsize=10, loc='lower right')
    plt.xlabel('Fraction of People in the Group')
    plt.title(f'Distribution of {column} by Classification', fontsize = 14)

plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/cat.png', dpi =␣
 ↪350)
plt.show()
```



We can find that all categorical attributes are binary. Also from the above stacked bar plot, we wonder if all not ckd happens in only one of the situation in all the binary attributes.

[15]:
```
results = {}
for i in cat_cols:
    uni = df[i].dropna().unique()
    results[i] = [
        round(len(df[(df[i] == uni[0]) & (df['classification'] == 'not ckd')])/
 ↪len(df[df[i] == uni[0]]),3),
        round(len(df[(df[i] == uni[0]) & (df['classification'] == 'ckd')])/
 ↪len(df[df[i] == uni[0]]),3),
        round(len(df[(df[i] == uni[1]) & (df['classification'] == 'not ckd')])/
 ↪len(df[df[i] == uni[1]]),3),
        round(len(df[(df[i] == uni[1]) & (df['classification'] == 'ckd')])/
 ↪len(df[df[i] == uni[1]]),3)
    ]
```

```
pd.DataFrame(results, index = ['notckdS1', 'ckdS1', 'notckdS2', 'ckdS2'])
```

[15]:

	rbc	pc	pcc	ba	htn	dm	cad	pe	ane	appet
notckdS1	0.701	0.544	0.412	0.39	0.00	0.000	0.407	0.461	0.44	0.47
ckdS1	0.299	0.456	0.588	0.61	1.00	1.000	0.593	0.539	0.56	0.53
notckdS2	0.000	0.000	0.000	0.00	0.59	0.567	0.000	0.000	0.00	0.00
ckdS2	1.000	1.000	1.000	1.00	0.41	0.433	1.000	1.000	1.00	1.00

0.1.4 (iv) Do some summary statistics

[16]:
```
df.shape
```

[16]: (400, 25)

Before preprocessing, we have 25 features and 400 data points(One feature is dropped because of colinearity)

[17]:
```
df.describe()
```

[17]:

	id	age	bp	sg	al	su
count	400.000000	391.000000	388.000000	353.000000	354.000000	351.000000
mean	199.500000	51.483376	76.469072	1.017408	1.016949	0.450142
std	115.614301	17.169714	13.683637	0.005717	1.352679	1.099191
min	0.000000	2.000000	50.000000	1.005000	0.000000	0.000000
25%	99.750000	42.000000	70.000000	1.010000	0.000000	0.000000
50%	199.500000	55.000000	80.000000	1.020000	0.000000	0.000000
75%	299.250000	64.500000	80.000000	1.020000	2.000000	0.000000
max	399.000000	90.000000	180.000000	1.025000	5.000000	5.000000

	bgr	bu	sc	sod	pot	pcv
count	356.000000	381.000000	383.000000	313.000000	312.000000	329.000000
mean	148.036517	57.425722	3.072454	137.528754	4.627244	38.884498
std	79.281714	50.503006	5.741126	10.408752	3.193904	8.990105
min	22.000000	1.500000	0.400000	4.500000	2.500000	9.000000
25%	99.000000	27.000000	0.900000	135.000000	3.800000	32.000000
50%	121.000000	42.000000	1.300000	138.000000	4.400000	40.000000
75%	163.000000	66.000000	2.800000	142.000000	4.900000	45.000000
max	490.000000	391.000000	76.000000	163.000000	47.000000	54.000000

	wc	rc
count	294.000000	269.000000
mean	8406.122449	4.707435
std	2944.474190	1.025323
min	2200.000000	2.100000
25%	6500.000000	3.900000
50%	8000.000000	4.800000
75%	9800.000000	5.400000

```
max     26400.000000     8.000000
```

0.2 Data Cleaning & Preparation

[18]:
```
for i in cat_cols:
    print(df[i].unique())
```

```
[nan 'normal' 'abnormal']
['normal' 'abnormal' nan]
['notpresent' 'present' nan]
['notpresent' 'present' nan]
['yes' 'no' nan]
['yes' 'no' nan]
['no' 'yes' nan]
['no' 'yes' nan]
['no' 'yes' nan]
['good' 'poor' nan]
```

[19]:
```
# replace nan with string (another category)
for i in cat_cols:
    df.loc[:,i] = df[i].replace(to_replace = {np.nan:'missing'})

for i in cat_cols:
    print(df[i].unique())
```

```
['missing' 'normal' 'abnormal']
['normal' 'abnormal' 'missing']
['notpresent' 'present' 'missing']
['notpresent' 'present' 'missing']
['yes' 'no' 'missing']
['yes' 'no' 'missing']
['no' 'yes' 'missing']
['no' 'yes' 'missing']
['no' 'yes' 'missing']
['good' 'poor' 'missing']
```

[20]:
```
# transfer the target variable from ['ckd','not ckd'] into 0s and 1s (easy for␣
 ↪future analysis)
warnings.filterwarnings("ignore")
df['classification'] = pd.DataFrame(df['classification'].
 ↪replace(to_replace={'ckd': 1, 'not ckd': 0}).reset_index(drop=True))
```

[21]:
```
# Check the missing values percentage for each attribute
df.isnull().sum()/len(df)
```

[21]:
```
id                0.0000
age               0.0225
bp                0.0300
```

```
sg                0.1175
al                0.1150
su                0.1225
rbc               0.0000
pc                0.0000
pcc               0.0000
ba                0.0000
bgr               0.1100
bu                0.0475
sc                0.0425
sod               0.2175
pot               0.2200
pcv               0.1775
wc                0.2650
rc                0.3275
htn               0.0000
dm                0.0000
cad               0.0000
appet             0.0000
pe                0.0000
ane               0.0000
classification    0.0000
dtype: float64
```

We can find the target variable doesn't contain missing values. And all attributes contain some missing values. And for the category variables, we can treat the missing values as another category.

0.3 Split and Preprocess the Dataset

[22]:
```
# keep random state the same for spliting and training the model
np.random.seed(42) # for reproducibility
random_states = list(np.random.randint(1, 10000, size = 10))
```

[23]:
```
X_train = []
X_val = []
X_test = []
y_train = []
y_val = []
y_test = []
X_test_withoutstd = []

X = df.loc[:, df.columns != 'classification']
y = df['classification']
# Define categorical and numerical features
cat_cols_noorder = ['rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'pe', 'ane',␣
 ↪'appet']
cat_cols_withorder = ['sg', 'al', 'su']
```

```
ordinal_cats = [[1.005,1.010,1.015,1.020,1.025,np.nan], [0,1,2,3,4,5,np.nan],␣
↪[0,1,2,3,4,5,np.nan]]

for i in range(len(random_states)):
    # use basic split to seperate out the test set
    X_train_unit, X_other_unit, y_train_unit, y_other_unit =␣
↪train_test_split(X, y, train_size = 0.6, random_state = random_states[i])
    X_val_unit, X_test_unit, y_val_unit, y_test_unit =␣
↪train_test_split(X_other_unit, y_other_unit, train_size = 0.5, random_state␣
↪= random_states[i])
    y_train.append(y_train_unit)
    y_val.append(y_val_unit)
    y_test.append(y_test_unit)

    # Preprocessing for the categorical data with no order -- OneHotEncoder
    # get the column names
    colnames = []
    for k in cat_cols_noorder:
        temp = [f'{k}_{value}' for value in X_train_unit[k].unique()]
        colnames.extend(temp)
    enc = OneHotEncoder(sparse_output = False, handle_unknown = 'ignore')
    X_train_pre_cat1 = pd.DataFrame(enc.
↪fit_transform(X_train_unit[cat_cols_noorder]), columns = colnames).
↪reset_index(drop=True)
    X_test_pre_cat1 = pd.DataFrame(enc.
↪transform(X_test_unit[cat_cols_noorder]), columns = colnames).
↪reset_index(drop=True)
    X_val_pre_cat1 = pd.DataFrame(enc.transform(X_val_unit[cat_cols_noorder]),␣
↪columns = colnames).reset_index(drop=True)

    # Preprocessing for the categorical data with order -- OrdinalEncoder
    ordinal = OrdinalEncoder(categories = ordinal_cats)
    X_train_pre_cat2 = pd.DataFrame(ordinal.
↪fit_transform(X_train_unit[cat_cols_withorder]), columns =␣
↪cat_cols_withorder).reset_index(drop=True)
    X_test_pre_cat2 = pd.DataFrame(ordinal.
↪transform(X_test_unit[cat_cols_withorder]), columns = cat_cols_withorder).
↪reset_index(drop=True)
    X_val_pre_cat2 = pd.DataFrame(ordinal.
↪transform(X_val_unit[cat_cols_withorder]), columns = cat_cols_withorder).
↪reset_index(drop=True)

    # Preprocessing for the continous variables -- StandardScalar
    scaler = StandardScaler()
```

```
    X_train_pre_num = pd.DataFrame(scaler.
↪fit_transform(X_train_unit[num_cols]), columns = num_cols).
↪reset_index(drop=True)
    X_test_pre_num = pd.DataFrame(scaler.transform(X_test_unit[num_cols]),␣
↪columns = num_cols).reset_index(drop=True)
    X_val_pre_num = pd.DataFrame(scaler.transform(X_val_unit[num_cols]),␣
↪columns = num_cols).reset_index(drop=True)

    # merge the table
    X_train_merge = pd.concat([X_train_pre_num, X_train_pre_cat2,␣
↪X_train_pre_cat1], axis = 1)
    X_val_merge = pd.concat([X_val_pre_num, X_val_pre_cat2, X_val_pre_cat1],␣
↪axis = 1)
    X_test_merge = pd.concat([X_test_pre_num, X_test_pre_cat2,␣
↪X_test_pre_cat1], axis = 1)

    # Apply StandardScalar to all the features
    X_train.append(pd.DataFrame(scaler.fit_transform(X_train_merge), columns =␣
↪X_train_merge.columns))
    X_val.append(pd.DataFrame(scaler.transform(X_val_merge), columns =␣
↪X_val_merge.columns))
    X_test.append(pd.DataFrame(scaler.transform(X_test_merge), columns =␣
↪X_test_merge.columns))

    # concat tables
    X_test_withoutstd.append(X_test_merge)
```

0.4 Choose ML model, Tune hyperparameter and Evaluate

0.4.1 (i) XGBoosting

[24]:
```
# Initialze empty variables
best_model_xgb = []
best_valscore_xgb = np.zeros(len(random_states))
test_score_xgb = np.zeros(len(random_states))
y_pred_xgb = []

for i in range(len(random_states)):
    # Define hyperparameter grid for tuning
    param_grid = {
        'max_depth': [1, 3, 5, 8],
        'n_estimators': [10000],
        'learning_rate': [0.03],
        'reg_alpha': [1e-4, 1e-3, 1e-2, 1e-1, 0e0],
        'reg_lambda': [1e-4, 1e-3, 1e-2, 1e-1, 0e0]
    }
```

```
    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []

    # train the model XGBoosting
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        clf = XGBClassifier(**params, early_stopping_rounds = 30, n_jobs = -1,␣
↪random_state = random_states[i])
        clf.fit(X_train[i], y_train[i], eval_set = [(X_val[i], y_val[i])],␣
↪verbose = False)
        models.append(clf)
        y_val_pred = clf.predict(X_val[i])
        val_score[p] = accuracy_score(y_val[i], y_val_pred)

    # get the best validation score
    best_valscore_xgb[i] = np.max(val_score)
    best_model_xgb.append(models[np.argmax(val_score)])

    # evaluate the model using test set (test score)
    y_test_pred = best_model_xgb[i].predict(X_test[i])
    test_score_xgb[i] = accuracy_score(y_test[i], y_test_pred)
    y_pred_xgb.append(y_test_pred)

    print('Random state: ', random_states[i])
    print('The best model parameters are: ', ParameterGrid(param_grid)[np.
↪argmax(val_score)])
    print('Best validation score: ', best_valscore_xgb[i])
    print('Test accuracy: ', test_score_xgb[i])
    print(' ')
```

```
Random state:  7271
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  861
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  0.9875
Test accuracy:  1.0

Random state:  5391
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  0.9875
Test accuracy:  0.9875
```

```
Random state:  5192
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  5735
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  6266
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  0.9875

Random state:  467
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  0.975

Random state:  4427
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  5579
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  8323
The best model parameters are:  {'reg_lambda': 0.0001, 'reg_alpha': 0.0001,
'n_estimators': 10000, 'max_depth': 1, 'learning_rate': 0.03}
Best validation score:  1.0
Test accuracy:  1.0
```

[25]:
```python
print('The average test accuracy for xgb is', test_score_xgb.mean())
print('The test accuracy standard deviation for xgb is', test_score_xgb.std())
y_test_flatten = [item for sublist in y_test for item in sublist]
```

```
y_pred_flatten = [item for sublist in y_pred_xgb for item in sublist] # get the␣
 ↪y_true and y_pred
overall_accuracy_xgb = accuracy_score(y_test_flatten, y_pred_flatten)
print('The OVERALL test accuracy for xgb is', overall_accuracy_xgb)
```

```
The average test accuracy for xgb is 0.9949999999999999
The test accuracy standard deviation for xgb is 0.008291561975888496
The OVERALL test accuracy for xgb is 0.995
```

When training with validation data, set a large n_estimators (e.g., 10000) and use early stopping to terminate the training once performance on the validation set stops improving. Here the test score stablized at 0.99, because we set large n_estimators and use early stopping to avoid overfitting. And here other parameters become less important with those two parameters settled down.

0.4.2 (ii) Random Forest

[26]:
```
# Before applying any other models, deal with missing values -- Multivariate␣
 ↪Imputation (IterativeImputer)
warnings.filterwarnings("ignore", category=ConvergenceWarning) # Suppress the␣
 ↪ConvergenceWarning
for i in range(len(random_states)):
    imputer = IterativeImputer(estimator = RandomForestRegressor(n_estimators =␣
 ↪50, n_jobs = -1), random_state = random_states[i])
    X_train[i] = pd.DataFrame(data = imputer.fit_transform(X_train[i]), columns␣
 ↪= X_train[i].columns)
    X_val[i] = pd.DataFrame(data = imputer.transform(X_val[i]), columns =␣
 ↪X_val[i].columns)
    X_test[i] = pd.DataFrame(data = imputer.transform(X_test[i]), columns =␣
 ↪X_test[i].columns)
```

[27]:
```
# Initialze empty variables
best_model_rf = []
best_valscore_rf = np.zeros(len(random_states))
test_score_rf = np.zeros(len(random_states))
y_pred_rf = []

for i in range(len(random_states)):
    # Define hyperparameter grid for tuning
    param_grid = {
        'n_estimators': [100],
        'max_depth': [1, 3, 5, 8, 13],
        'max_features': [1, 3, 5, 8, 13, 18]
    }

    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []
```

```
    # train the model random forest
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        clf = RandomForestClassifier(**params, n_jobs = -1, random_state =␣
↪random_states[i])
        clf.fit(X_train[i], y_train[i])
        models.append(clf)
        y_val_pred = clf.predict(X_val[i])
        val_score[p] = accuracy_score(y_val[i], y_val_pred)

    # get the best validation score
    best_valscore_rf[i] = np.max(val_score)
    best_model_rf.append(models[np.argmax(val_score)])

    # evaluate the model using test set (test score)
    y_test_pred = best_model_rf[i].predict(X_test[i])
    test_score_rf[i] = accuracy_score(y_test[i], y_test_pred)
    y_pred_rf.append(y_test_pred)

    print('Random state: ', random_states[i])
    print('The best model parameters are: ', ParameterGrid(param_grid)[np.
↪argmax(val_score)])
    print('Best validation score: ', best_valscore_rf[i])
    print('Test accuracy: ', test_score_rf[i])
    print(' ')
```

```
Random state:  7271
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 1}
Best validation score:  1.0
Test accuracy:  0.95

Random state:  861
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 3}
Best validation score:  1.0
Test accuracy:  0.9875

Random state:  5391
The best model parameters are:  {'n_estimators': 100, 'max_features': 13,
'max_depth': 3}
Best validation score:  1.0
Test accuracy:  0.9875

Random state:  5192
The best model parameters are:  {'n_estimators': 100, 'max_features': 5,
'max_depth': 1}
```

```
Best validation score:  1.0
Test accuracy:  0.9375

Random state:  5735
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 3}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  6266
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 1}
Best validation score:  1.0
Test accuracy:  0.9625

Random state:  467
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 1}
Best validation score:  1.0
Test accuracy:  0.9625

Random state:  4427
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 8}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  5579
The best model parameters are:  {'n_estimators': 100, 'max_features': 5,
'max_depth': 3}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  8323
The best model parameters are:  {'n_estimators': 100, 'max_features': 3,
'max_depth': 8}
Best validation score:  1.0
Test accuracy:  1.0
```

[28]:
```
print('The average test accuracy for rf is', test_score_rf.mean())
print('The test accuracy standard deviation for rf is', test_score_rf.std())
y_pred_flatten = [item for sublist in y_pred_rf for item in sublist]
overall_accuracy_rf = accuracy_score(y_test_flatten, y_pred_flatten)
print('The OVERALL test accuracy for rf is', overall_accuracy_rf)
```

```
The average test accuracy for rf is 0.97875
The test accuracy standard deviation for rf is 0.022395591083961153
```

```
The OVERALL test accuracy for rf is 0.97875
```

0.4.3 (iii) SVM

[29]:
```
# Initialze empty variables
best_model_svm = []
best_valscore_svm = np.zeros(len(random_states))
test_score_svm = np.zeros(len(random_states))
y_pred_svm = []

for i in range(len(random_states)):
    # Define hyperparameter grid for tuning
    param_grid = {
        'C': [1e-1, 1e0, 1e1, 1e2, 1e3, 1e4],
        'gamma':[1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1]
    }

    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []
    # train the model SVM
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        clf = SVC(**params, probability = True, random_state =␣
 ↪random_states[i]) # use SVC for classification problem
        clf.fit(X_train[i], y_train[i])
        models.append(clf)
        y_val_pred = clf.predict(X_val[i])
        val_score[p] = accuracy_score(y_val[i], y_val_pred)

    # get the best validation score
    best_valscore_svm[i] = np.max(val_score)
    best_model_svm.append(models[np.argmax(val_score)])

    # evaluate the model using test set (test score)
    y_test_pred = best_model_svm[i].predict(X_test[i])
    test_score_svm[i] = accuracy_score(y_test[i], y_test_pred)
    y_pred_svm.append(y_test_pred)

    print('Random state: ', random_states[i])
    print('The best model parameters are: ', ParameterGrid(param_grid)[np.
 ↪argmax(val_score)])
    print('Best validation score: ', best_valscore_svm[i])
    print('Test accuracy: ', test_score_svm[i])
    print(' ')
```

```
Random state:  7271
The best model parameters are:  {'gamma': 0.01, 'C': 1.0}
Best validation score:  1.0
```

```
Test accuracy:  0.9625

Random state:  861
The best model parameters are:  {'gamma': 0.1, 'C': 1.0}
Best validation score:  0.9875
Test accuracy:  0.9625

Random state:  5391
The best model parameters are:  {'gamma': 0.01, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  0.9875

Random state:  5192
The best model parameters are:  {'gamma': 0.01, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  5735
The best model parameters are:  {'gamma': 0.01, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  6266
The best model parameters are:  {'gamma': 0.01, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  0.975

Random state:  467
The best model parameters are:  {'gamma': 0.1, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  4427
The best model parameters are:  {'gamma': 0.1, 'C': 0.1}
Best validation score:  1.0
Test accuracy:  0.925

Random state:  5579
The best model parameters are:  {'gamma': 0.1, 'C': 1.0}
Best validation score:  1.0
Test accuracy:  0.9875

Random state:  8323
The best model parameters are:  {'gamma': 0.01, 'C': 100.0}
Best validation score:  1.0
Test accuracy:  1.0
```

[30]:
```
print('The average test accuracy for svm is', test_score_svm.mean())
print('The test accuracy standard deviation for svm is', test_score_svm.std())
y_pred_flatten = [item for sublist in y_pred_svm for item in sublist]
overall_accuracy_svm = accuracy_score(y_test_flatten, y_pred_flatten)
print('The OVERALL test accuracy for rf is', overall_accuracy_svm)
```

```
The average test accuracy for svm is 0.9800000000000001
The test accuracy standard deviation for svm is 0.023184046238739247
The OVERALL test accuracy for rf is 0.98
```

0.4.4 (iv) Logistic Regression with Elastic Net Regularization

[31]:
```
# Initialze empty variables
best_model_logR = []
best_valscore_logR = np.zeros(len(random_states))
test_score_logR = np.zeros(len(random_states))
y_pred_logR = []

for i in range(len(random_states)):
    # Define hyperparameter grid for tuning
    param_grid = {
        'penalty': ['elasticnet'],
        'solver': ['saga'],
        'l1_ratio': np.linspace(0,1,8),
        'C': [0.01, 0.05, 0.1, 0.5, 1, 1.5, 2]
    }

    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []

    # train the model
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        clf = LogisticRegression(**params, random_state = random_states[i])
        clf.fit(X_train[i], y_train[i])
        models.append(clf)
        y_val_pred = clf.predict(X_val[i])
        val_score[p] = accuracy_score(y_val[i], y_val_pred)

    # get the best validation score
    best_valscore_logR[i] = np.max(val_score)
    best_model_logR.append(models[np.argmax(val_score)])

    # evaluate the model using test set (test score)
    y_test_pred = best_model_logR[i].predict(X_test[i])
    test_score_logR[i] = accuracy_score(y_test[i], y_test_pred)
    y_pred_logR.append(y_test_pred)
```

```
    print('Random state: ', random_states[i])
    print('The best model parameters are: ', ParameterGrid(param_grid)[np.
↪argmax(val_score)])
    print('Best validation score: ', best_valscore_logR[i])
    print('Test accuracy: ', test_score_logR[i])
    print(' ')
```

```
Random state:  7271
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.01}
Best validation score:  1.0
Test accuracy:  0.95

Random state:  861
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.05}
Best validation score:  0.9875
Test accuracy:  0.9875

Random state:  5391
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.05}
Best validation score:  0.9875
Test accuracy:  0.975

Random state:  5192
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.01}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  5735
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.05}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  6266
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.01}
Best validation score:  1.0
Test accuracy:  0.95

Random state:  467
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.05}
Best validation score:  1.0
```

```
Test accuracy:  0.95

Random state:  4427
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.14285714285714285, 'C': 0.01}
Best validation score:  1.0
Test accuracy:  0.975

Random state:  5579
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.1}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  8323
The best model parameters are:  {'solver': 'saga', 'penalty': 'elasticnet',
'l1_ratio': 0.0, 'C': 0.01}
Best validation score:  0.9875
Test accuracy:  0.9875
```

[32]:
```
print('The average test accuracy for Logistic Regression with Elastic Net␣
 ↪Regularization is', test_score_logR.mean())
print('The test accuracy standard deviation for Logistic Regression with␣
 ↪Elastic Net Regularization is', test_score_logR.std())
y_pred_flatten = [item for sublist in y_pred_logR for item in sublist]
overall_accuracy_logR = accuracy_score(y_test_flatten, y_pred_flatten)
print('The OVERALL test accuracy for Logistic Regression with Elastic Net␣
 ↪Regularization is', overall_accuracy_logR)
```

```
The average test accuracy for Logistic Regression with Elastic Net
Regularization is 0.9775
The test accuracy standard deviation for Logistic Regression with Elastic Net
Regularization is 0.020000000000000025
The OVERALL test accuracy for Logistic Regression with Elastic Net
Regularization is 0.9775
```

0.4.5 (v) KNN

[33]:
```
# Initialze empty variables
best_model_knn = []
best_valscore_knn = np.zeros(len(random_states))
test_score_knn = np.zeros(len(random_states))
y_pred_knn = []

for i in range(len(random_states)):
    # Define hyperparameter grid for tuning
    param_grid = {
```

```
        'n_neighbors': [1, 3, 5, 10],  # Number of neighbors
        'weights': ['uniform', 'distance'],  # Weighting scheme
        'metric': ['euclidean', 'manhattan', 'minkowski'],  # Distance metrics
    }

    val_score = np.zeros(len(ParameterGrid(param_grid)))
    models = []

    # train the model
    for p in range(len(ParameterGrid(param_grid))):
        params = ParameterGrid(param_grid)[p]
        clf = KNeighborsClassifier(**params)
        clf.fit(X_train[i], y_train[i])
        models.append(clf)
        y_val_pred = clf.predict(X_val[i])
        val_score[p] = accuracy_score(y_val[i], y_val_pred)

    # get the best validation score
    best_valscore_knn[i] = np.max(val_score)
    best_model_knn.append(models[np.argmax(val_score)])

    # evaluate the model using test set (test score)
    y_test_pred = best_model_knn[i].predict(X_test[i])
    test_score_knn[i] = accuracy_score(y_test[i], y_test_pred)
    y_pred_knn.append(y_test_pred)

    print('Random state: ', random_states[i])
    print('The best model parameters are: ', ParameterGrid(param_grid)[np.
↪argmax(val_score)])
    print('Best validation score: ', best_valscore_knn[i])
    print('Test accuracy: ', test_score_knn[i])
    print(' ')
```

```
Random state:  7271
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 3,
'metric': 'euclidean'}
Best validation score:  1.0
Test accuracy:  0.975

Random state:  861
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  0.9875
Test accuracy:  0.9875

Random state:  5391
The best model parameters are:  {'weights': 'distance', 'n_neighbors': 10,
'metric': 'euclidean'}
```

```
Best validation score:  0.975
Test accuracy:  0.975

Random state:  5192
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  0.9875
Test accuracy:  0.9875

Random state:  5735
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  1.0
Test accuracy:  0.9625

Random state:  6266
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  0.975
Test accuracy:  1.0

Random state:  467
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  1.0
Test accuracy:  0.9625

Random state:  4427
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  1.0
Test accuracy:  0.975

Random state:  5579
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'manhattan'}
Best validation score:  1.0
Test accuracy:  1.0

Random state:  8323
The best model parameters are:  {'weights': 'uniform', 'n_neighbors': 1,
'metric': 'euclidean'}
Best validation score:  0.9875
Test accuracy:  0.9875
```

[34]:
```
print('The average test accuracy for KNN is', test_score_knn.mean())
print('The test accuracy standard deviation for KNN is', test_score_knn.std())
y_pred_flatten = [item for sublist in y_pred_knn for item in sublist]
overall_accuracy_knn = accuracy_score(y_test_flatten, y_pred_flatten)
print('The OVERALL test accuracy for Logistic KNN is', overall_accuracy_knn)
```

```
The average test accuracy for KNN is 0.98125
The test accuracy standard deviation for KNN is 0.0128086884574495
The OVERALL test accuracy for Logistic KNN is 0.98125
```

0.4.6 Summary of result

Since we have balanced dataset (0.6 percentage and 0.4 percentage for binary target variable), we can use the accuracy score as our baseline score. And the strategy of basaline accuracy score is to take the majority of one class divide by the total number of data points. Here I use y_test from all 10 different random states.

[35]:
```
x = sum(y_test_flatten)/len(y_test_flatten)
if x >= 0.5:
    baseline_score = x
else:
    baseline_score = 1 - x
print(baseline_score)
```

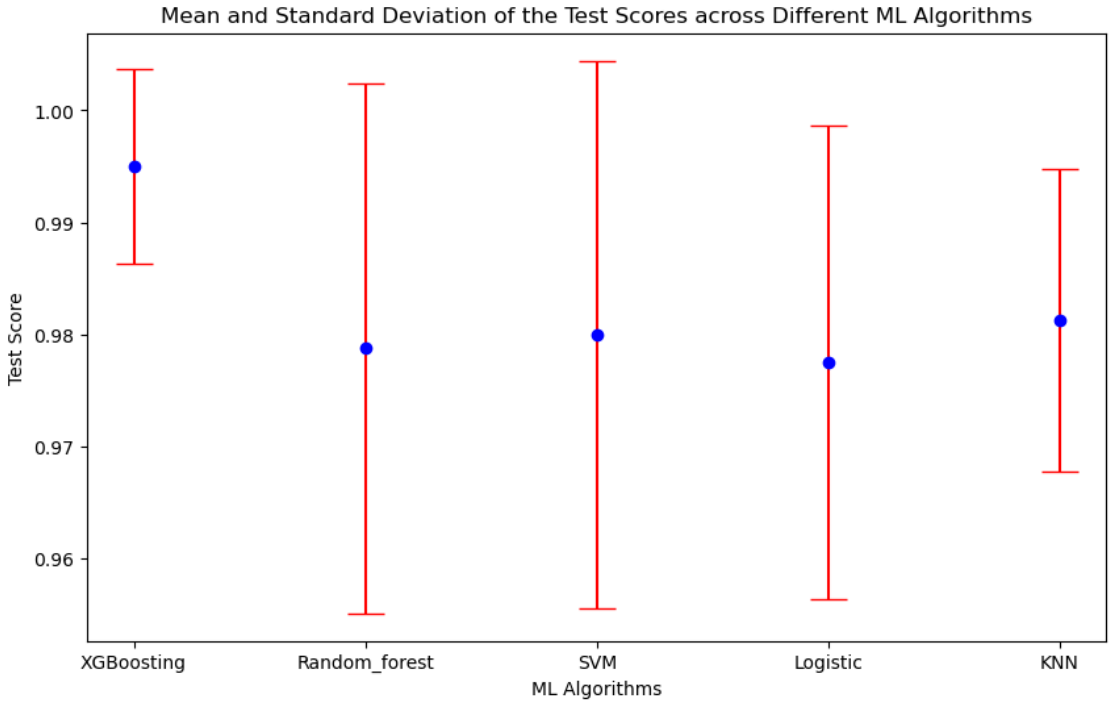
```
0.63125
```

[36]:
```
test_score_sum = pd.DataFrame({'XGBoosting': test_score_xgb,
                               'Random_forest': test_score_rf,
                               'SVM': test_score_svm,
                               'Logistic': test_score_logR,
                               'KNN': test_score_knn})
print(test_score_sum)
```

```
   XGBoosting  Random_forest     SVM  Logistic     KNN
0      1.0000         0.9500  0.9625    0.9500  0.9750
1      1.0000         0.9875  0.9625    0.9875  0.9875
2      0.9875         0.9875  0.9875    0.9750  0.9750
3      1.0000         0.9375  1.0000    1.0000  0.9875
4      1.0000         1.0000  1.0000    1.0000  0.9625
5      0.9875         0.9625  0.9750    0.9500  1.0000
6      0.9750         0.9625  1.0000    0.9500  0.9625
7      1.0000         1.0000  0.9250    0.9750  0.9750
8      1.0000         1.0000  0.9875    1.0000  1.0000
9      1.0000         1.0000  1.0000    0.9875  0.9875
```

[37]:
```
test_mean = test_score_sum.mean()
test_std = test_score_sum.std()
# plot means and standard deviations of test scores
plt.figure(figsize = (10,6))
```

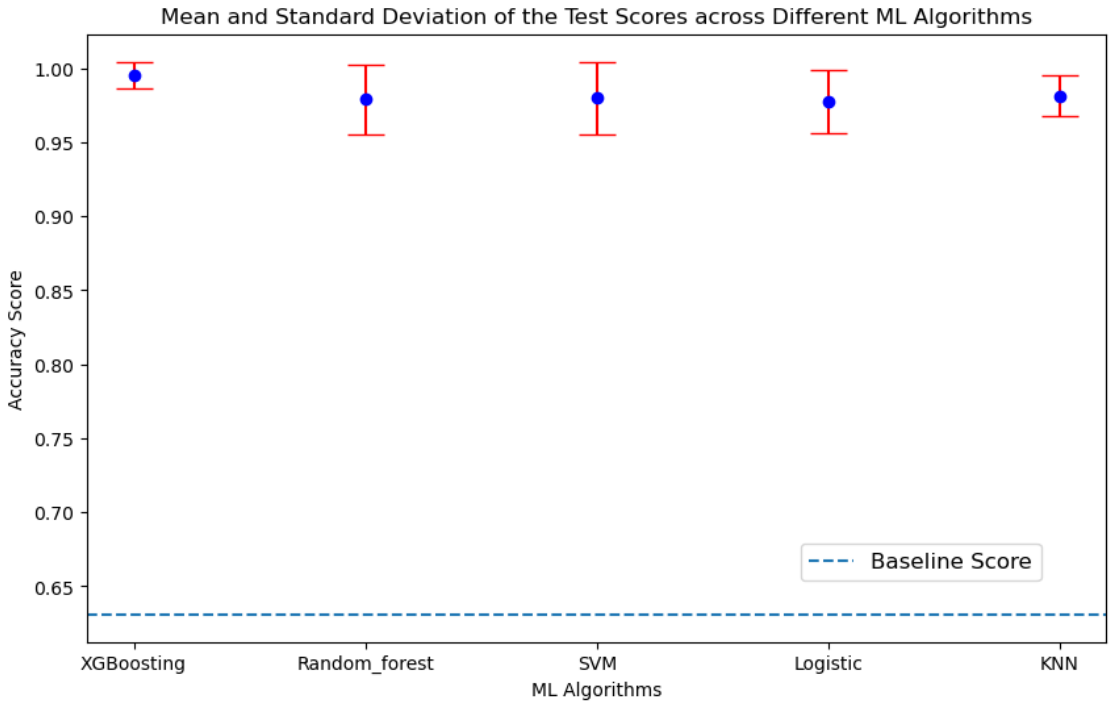
```
plt.errorbar(x = test_mean.index, y = test_mean, yerr = test_std, fmt␣
 ↪='o',color='blue', ecolor='red', capsize=10)
plt.xlabel('ML Algorithms')
plt.ylabel('Test Score')
plt.title('Mean and Standard Deviation of the Test Scores across Different ML␣
 ↪Algorithms')
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/
 ↪compare_result_ML(easy).png', dpi = 350)
plt.show()
print(test_score_sum.mean())
```




```
XGBoosting       0.99500
Random_forest    0.97875
SVM              0.98000
Logistic         0.97750
KNN              0.98125
dtype: float64
```

[38]:
```
# add a baseline on the plot
plt.figure(figsize = (10,6))
plt.errorbar(x = test_mean.index, y = test_mean, yerr = test_std, fmt␣
 ↪='o',color='blue', ecolor='red', capsize=10)
plt.xlabel('ML Algorithms')
plt.ylabel('Accuracy Score')
```

```
plt.title('Mean and Standard Deviation of the Test Scores across Different ML␣
 ↪Algorithms')
plt.axhline(baseline_score, linestyle='--', label='Baseline Score')
plt.legend(loc= (0.7, 0.1), fontsize = 12)
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/
 ↪compare_result_ML.png', dpi = 350)
plt.show()
print('Baseline_score: ', baseline_score)
```



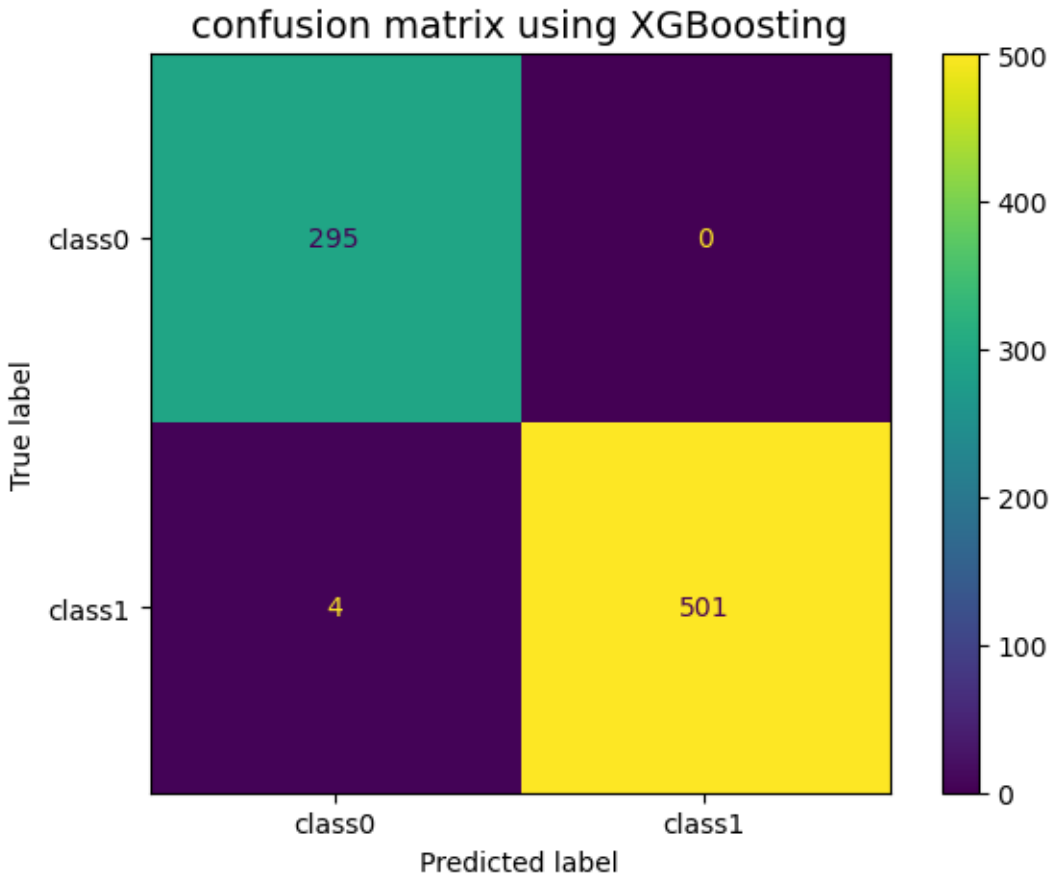

```
Baseline_score:  0.63125
```

0.5 Model Inspectation

Here we choose to use XGBoosting model because it has the highest accuracy score, and then we use the y_pred from this model to do the model inspection. ### (i) confusion matrix

[39]:
```
# confusion matrix using XGBoosting
cm = confusion_matrix(y_test_flatten, [item for sublist in y_pred_xgb for item␣
 ↪in sublist])
disp = ConfusionMatrixDisplay(cm, display_labels = ['class0', 'class1'])
fig, ax = plt.subplots(figsize = (7,5))
disp.plot(ax=ax)
plt.title('confusion matrix using XGBoosting', fontsize = 14)
```

```
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/
  ↪confusion_matrix_xgb.png', dpi = 350)
plt.show()

# calculate metrics
recall = cm[1][1]/sum(cm[1])
precision = cm[1][1]/(cm[1][1] + cm[0][1])
accuracy = (cm[0][0] + cm[1][1])/sum(sum(cm))
f1 = 2*precision*recall/(precision + recall)

print('recall: ', recall)
print('precision: ', precision)
print('accuracy: ', accuracy)
print('f1 score: ', f1)
```




```
recall:  0.9920792079207921
precision:  1.0
accuracy:  0.995
f1 score:  0.9960238568588469
```

Next, we want to investigate some bias:

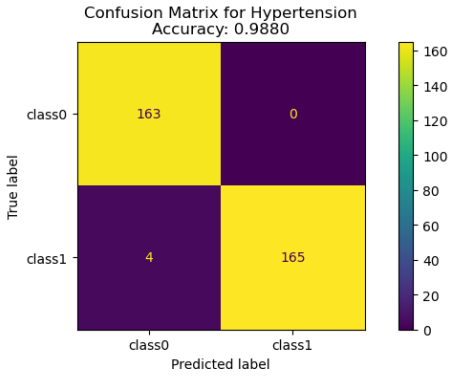
[40]:
```
finaldf = []
for i in range(len(random_states)):
    combinecol = pd.concat([pd.DataFrame(X_test[i]).reset_index(drop=True),
                            pd.DataFrame(y_test[i]).reset_index(drop=True),
                            pd.DataFrame(y_pred_xgb[i]).
 ↪reset_index(drop=True)], axis = 1)
    colnames = list(X_test[i].columns) + ['y_true', 'y_pred']
    combinecol.columns = colnames
    finaldf.append(combinecol)
```

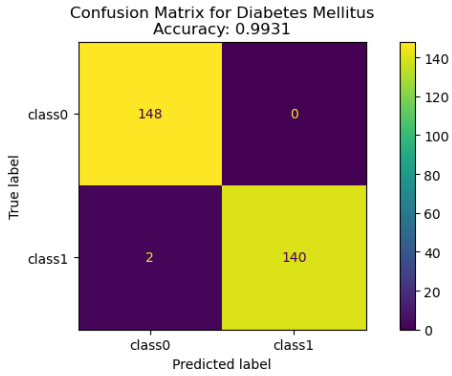
[41]:
```
df1 = []
df2 = []
df3 = []
df4 = []
df5 = []
df6 = []
df7 = []
df8 = []
df9 = []
df10 = []

for i in range(len(random_states)):
    # Data and metrics for hypertension people
    temp_data1 = finaldf[i][X_test_withoutstd[i]['htn_yes'] == 1][['y_true',␣
 ↪'y_pred']]
    df1.append(temp_data1)

    # Data and metrics for diabetes mellitus people
    temp_data2 = finaldf[i][X_test_withoutstd[i]['dm_yes'] == 1][['y_true',␣
 ↪'y_pred']]
    df2.append(temp_data2)

    # Data and metrics for coronary artery disease people
    temp_data3 = finaldf[i][X_test_withoutstd[i]['cad_yes'] == 1][['y_true',␣
 ↪'y_pred']]
    df3.append(temp_data3)

    # Data and metrics for appetite people
    temp_data4 = finaldf[i][X_test_withoutstd[i]['appet_good'] == 1][['y_true',␣
 ↪'y_pred']]
    df4.append(temp_data4)

    # Data and metrics for pedal edema people
    temp_data5 = finaldf[i][X_test_withoutstd[i]['pe_yes'] == 1][['y_true',␣
 ↪'y_pred']]
```

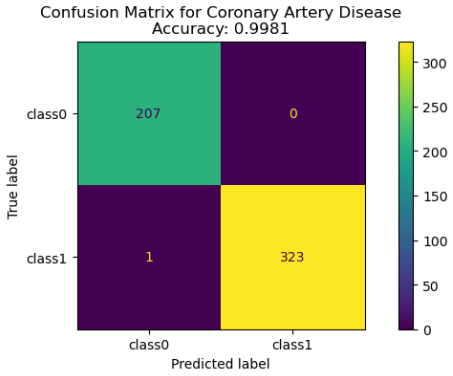
```
    df5.append(temp_data5)

    # Data and metrics for anemia people
    temp_data6 = finaldf[i][X_test_withoutstd[i]['ane_yes'] == 1][['y_true',␣
↪'y_pred']]
    df6.append(temp_data6)

    # Data and metrics for red blood cells people
    temp_data7 = finaldf[i][X_test_withoutstd[i]['rbc_normal'] == 1][['y_true',␣
↪'y_pred']]
    df7.append(temp_data7)

    # Data and metrics for pus cell people
    temp_data8 = finaldf[i][X_test_withoutstd[i]['pc_normal'] == 1][['y_true',␣
↪'y_pred']]
    df8.append(temp_data8)

    # Data and metrics for pus cell clumps people
    temp_data9 = finaldf[i][X_test_withoutstd[i]['pcc_present'] ==␣
↪1][['y_true', 'y_pred']]
    df9.append(temp_data9)

    # Data and metrics for becteria people
    temp_data10 = finaldf[i][X_test_withoutstd[i]['ba_present'] ==␣
↪1][['y_true', 'y_pred']]
    df10.append(temp_data10)

df1 = pd.concat(df1, axis = 0)
df2 = pd.concat(df2, axis = 0)
df3 = pd.concat(df3, axis = 0)
df4 = pd.concat(df4, axis = 0)
df5 = pd.concat(df5, axis = 0)
df6 = pd.concat(df6, axis = 0)
df7 = pd.concat(df7, axis = 0)
df8 = pd.concat(df8, axis = 0)
df9 = pd.concat(df9, axis = 0)
df10 = pd.concat(df10, axis = 0)

ac1 = accuracy_score(df1['y_true'], df1['y_pred'])
cm1 = confusion_matrix(df1['y_true'], df1['y_pred'])
ac2 = accuracy_score(df2['y_true'], df2['y_pred'])
cm2 = confusion_matrix(df2['y_true'], df2['y_pred'])
ac3 = accuracy_score(df3['y_true'], df3['y_pred'])
cm3 = confusion_matrix(df3['y_true'], df3['y_pred'])
ac4 = accuracy_score(df4['y_true'], df4['y_pred'])
cm4 = confusion_matrix(df4['y_true'], df4['y_pred'])
ac5 = accuracy_score(df5['y_true'], df5['y_pred'])
```

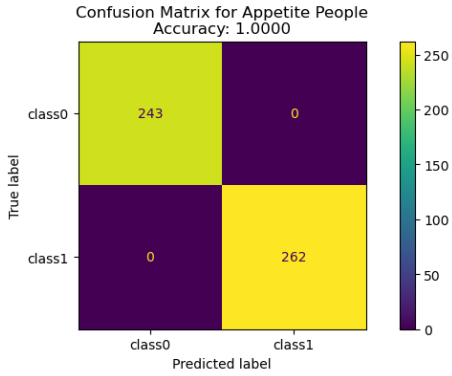
```
cm5 = confusion_matrix(df5['y_true'], df5['y_pred'])
ac6 = accuracy_score(df6['y_true'], df6['y_pred'])
cm6 = confusion_matrix(df6['y_true'], df6['y_pred'])
ac7 = accuracy_score(df7['y_true'], df7['y_pred'])
cm7 = confusion_matrix(df7['y_true'], df7['y_pred'])
ac8 = accuracy_score(df8['y_true'], df8['y_pred'])
cm8 = confusion_matrix(df8['y_true'], df8['y_pred'])
ac9 = accuracy_score(df9['y_true'], df9['y_pred'])
cm9 = confusion_matrix(df9['y_true'], df9['y_pred'])
ac10 = accuracy_score(df10['y_true'], df10['y_pred'])
cm10 = confusion_matrix(df10['y_true'], df10['y_pred'])

# Plots
# Create a grid for the plots (5 rows x 2 columns)
fig, axes = plt.subplots(5, 2, figsize=(14, 20))  # Adjust figsize for better␣
 ↪visualization

# Plot for hypertension people
disp1 = ConfusionMatrixDisplay(cm1, display_labels=['class0', 'class1'])
disp1.plot(ax=axes[0, 0], values_format='d')
axes[0, 0].set_title(f'Confusion Matrix for Hypertension\nAccuracy: {ac1:.4f}',␣
 ↪fontsize=12)

# Plot for diabetes mellitus people
disp2 = ConfusionMatrixDisplay(cm2, display_labels=['class0', 'class1'])
disp2.plot(ax=axes[0, 1], values_format='d')
axes[0, 1].set_title(f'Confusion Matrix for Diabetes Mellitus\nAccuracy: {ac2:.
 ↪4f}', fontsize=12)

# Plot for coronary artery disease people
disp3 = ConfusionMatrixDisplay(cm3, display_labels=['class0', 'class1'])
disp3.plot(ax=axes[1, 0], values_format='d')
axes[1, 0].set_title(f'Confusion Matrix for Coronary Artery Disease\nAccuracy:␣
 ↪{ac3:.4f}', fontsize=12)

# Plot for appetite people
disp4 = ConfusionMatrixDisplay(cm4, display_labels=['class0', 'class1'])
disp4.plot(ax=axes[1, 1], values_format='d')
axes[1, 1].set_title(f'Confusion Matrix for Appetite People\nAccuracy: {ac4:.
 ↪4f}', fontsize=12)

# Plot for pedal edema people
disp5 = ConfusionMatrixDisplay(cm5, display_labels=['class0', 'class1'])
disp5.plot(ax=axes[2, 0], values_format='d')
axes[2, 0].set_title(f'Confusion Matrix for Pedal Edema\nAccuracy: {ac5:.4f}',␣
 ↪fontsize=12)
```

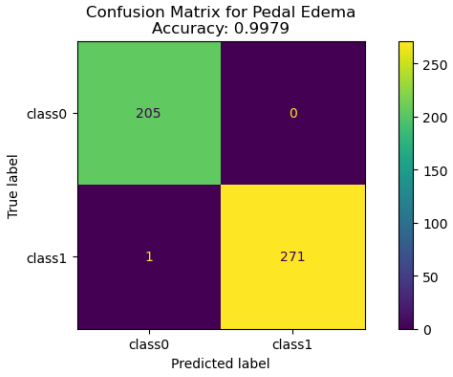
```
# Plot for anemia people
disp6 = ConfusionMatrixDisplay(cm6, display_labels=['class0', 'class1'])
disp6.plot(ax=axes[2, 1], values_format='d')
axes[2, 1].set_title(f'Confusion Matrix for Anemia\nAccuracy: {ac6:.4f}',␣
 ↪fontsize=12)

# Plot for red blood cells people
disp7 = ConfusionMatrixDisplay(cm7, display_labels=['class0', 'class1'])
disp7.plot(ax=axes[3, 0], values_format='d')
axes[3, 0].set_title(f'Confusion Matrix for Red Blood Cells\nAccuracy: {ac7:.
 ↪4f}', fontsize=12)

# Plot for pus cell people
disp8 = ConfusionMatrixDisplay(cm8, display_labels=['class0', 'class1'])
disp8.plot(ax=axes[3, 1], values_format='d')
axes[3, 1].set_title(f'Confusion Matrix for Pus Cells\nAccuracy: {ac8:.4f}',␣
 ↪fontsize=12)

# Plot for pus cell clumps people
disp9 = ConfusionMatrixDisplay(cm9, display_labels=['class0', 'class1'])
disp9.plot(ax=axes[4, 0], values_format='d')
axes[4, 0].set_title(f'Confusion Matrix for Pus Cell Clumps\nAccuracy: {ac9:.
 ↪4f}', fontsize=12)

# Plot for bacteria people
disp10 = ConfusionMatrixDisplay(cm10, display_labels=['class0', 'class1'])
disp10.plot(ax=axes[4, 1], values_format='d')
axes[4, 1].set_title(f'Confusion Matrix for Bacteria\nAccuracy: {ac10:.4f}',␣
 ↪fontsize=12)

# Adjust layout to avoid overlapping
plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/
 ↪confusion_matrix_forbias.png', dpi = 350)
plt.show()
```

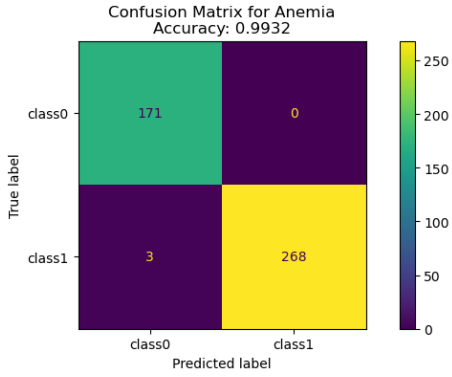

Confusion Matrix for Hypertension
Accuracy: 0.9880
class0
class1
True label
163
0
4
165
class0
class1
Predicted label
160
140
120
100
80
60
40
20
0

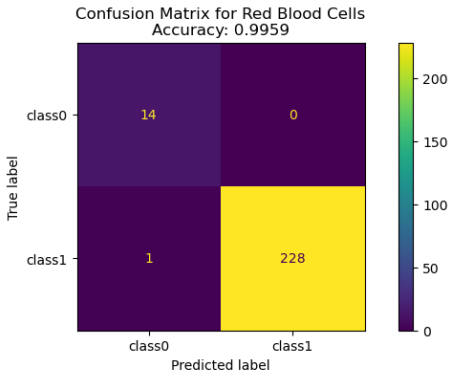

Confusion Matrix for Diabetes Mellitus
Accuracy: 0.9931
class0
class1
True label
148
0
2
140
class0
class1
Predicted label
140
120
100
80
60
40
20
0

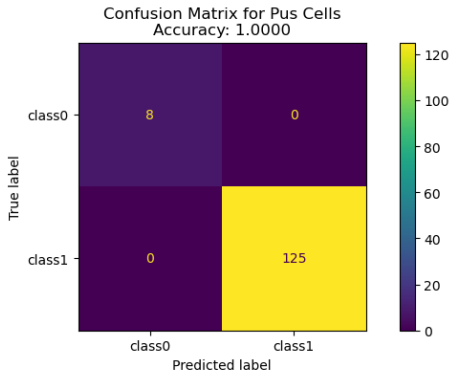

Confusion Matrix for Coronary Artery Disease
Accuracy: 0.9981
class0
class1
True label
207
0
1
323
class0
class1
Predicted label
300
250
200
150
100
50
0

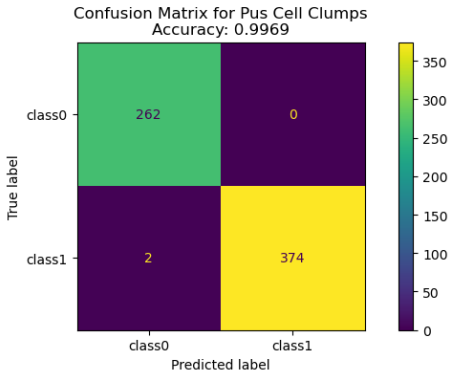

Confusion Matrix for Appetite People
Accuracy: 1.0000
class0
class1
True label
243
0
0
262
class0
class1
Predicted label
250
200
150
100
50
0

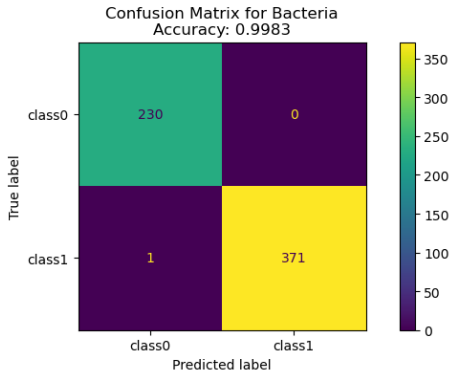

Confusion Matrix for Pedal Edema
Accuracy: 0.9979
class0
class1
True label
205
0
1
271
class0
class1
Predicted label
250
200
150
100
50
0


Confusion Matrix for Anemia
Accuracy: 0.9932
class0
class1
True label
171
0
3
268
class0
class1
Predicted label
250
200
150
100
50
0


Confusion Matrix for Red Blood Cells
Accuracy: 0.9959
class0
class1
True label
14
0
1
228
class0
class1
Predicted label
200
150
100
50
0


Confusion Matrix for Pus Cells
Accuracy: 1.0000
class0
class1
True label
8
0
0
125
class0
class1
Predicted label
120
100
80
60
40
20
0


Confusion Matrix for Pus Cell Clumps
Accuracy: 0.9969
class0
class1
True label
262
0
2
374
class0
class1
Predicted label
350
300
250
200
150
100
50
0

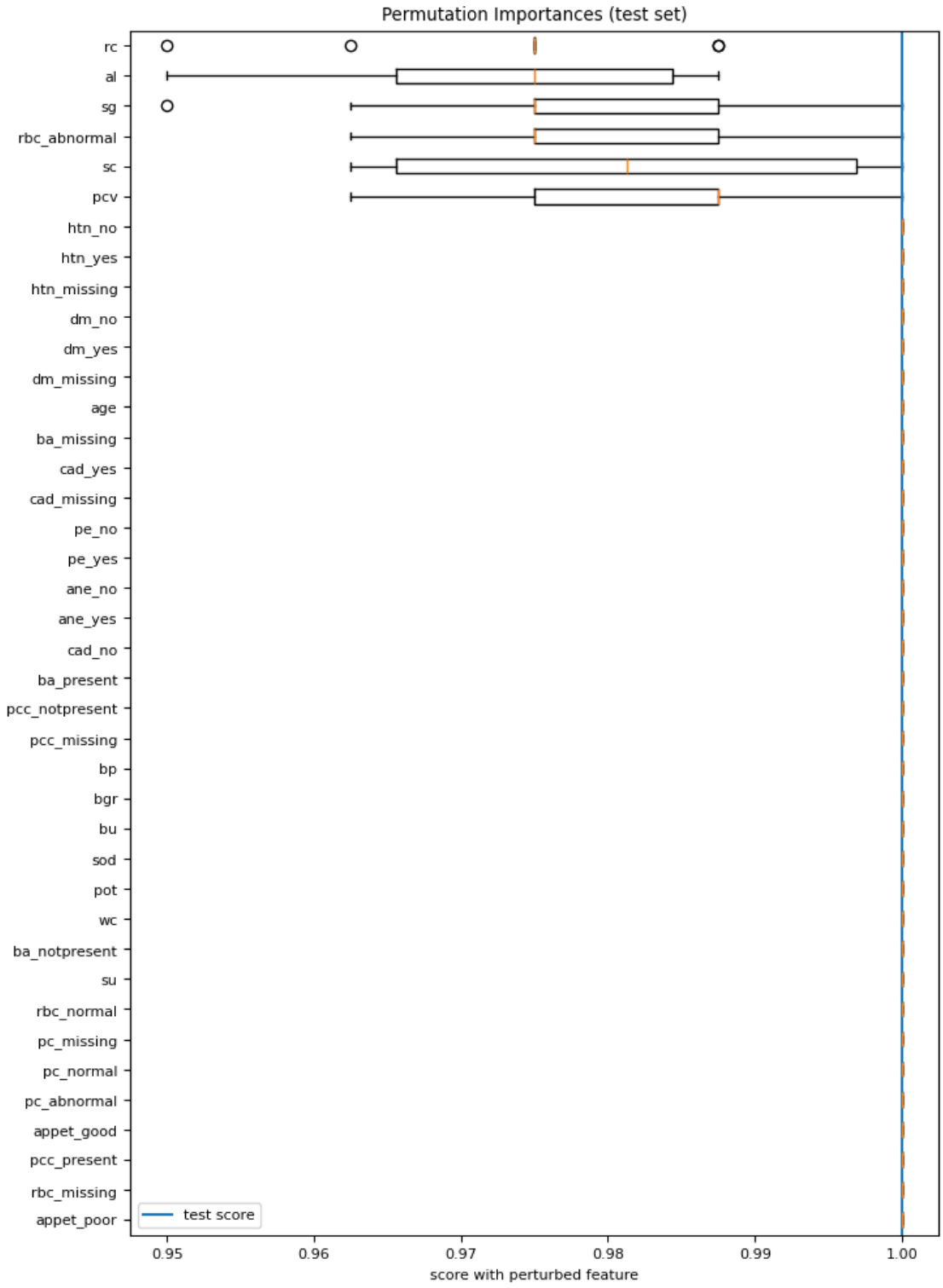

Confusion Matrix for Bacteria
Accuracy: 0.9983
class0
class1
True label
230
0
1
371
class0
class1
Predicted label
350
300
250
200
150
100
50
0

0.5.1 (ii) feature importance

Global importance

method1:

[28]:
```
np.random.seed(42) # for reproducibility
nr_runs = 10
best_index = np.argmax(test_score_xgb)
feature_names = X_test[best_index].columns
scores = np.zeros([len(feature_names), nr_runs])

for i in range(len(feature_names)):
    acc_scores = []
    for j in range(nr_runs):
        X_test_shuffled = X_test[best_index].copy()
        X_test_shuffled[feature_names[i]] = np.random.
 ↪permutation(X_test[best_index][feature_names[i]].values)
        acc_scores.append(best_model_xgb[best_index].score(X_test_shuffled,␣
 ↪y_test[best_index]))
    scores[i] = acc_scores

sorted_indcs = np.argsort(np.mean(scores, axis = 1))
plt.rcParams.update({'font.size': 8})
plt.figure(figsize=(8,12))
plt.boxplot(scores[sorted_indcs].T,tick_labels = feature_names[sorted_indcs],␣
 ↪vert=False)
plt.title("Permutation Importances (test set)")
plt.xlabel('score with perturbed feature')
plt.axvline(test_score_xgb[best_index], label='test score')
plt.gca().invert_yaxis()
plt.legend()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/permutation_imp.
 ↪png', dpi = 350)
plt.show()
```

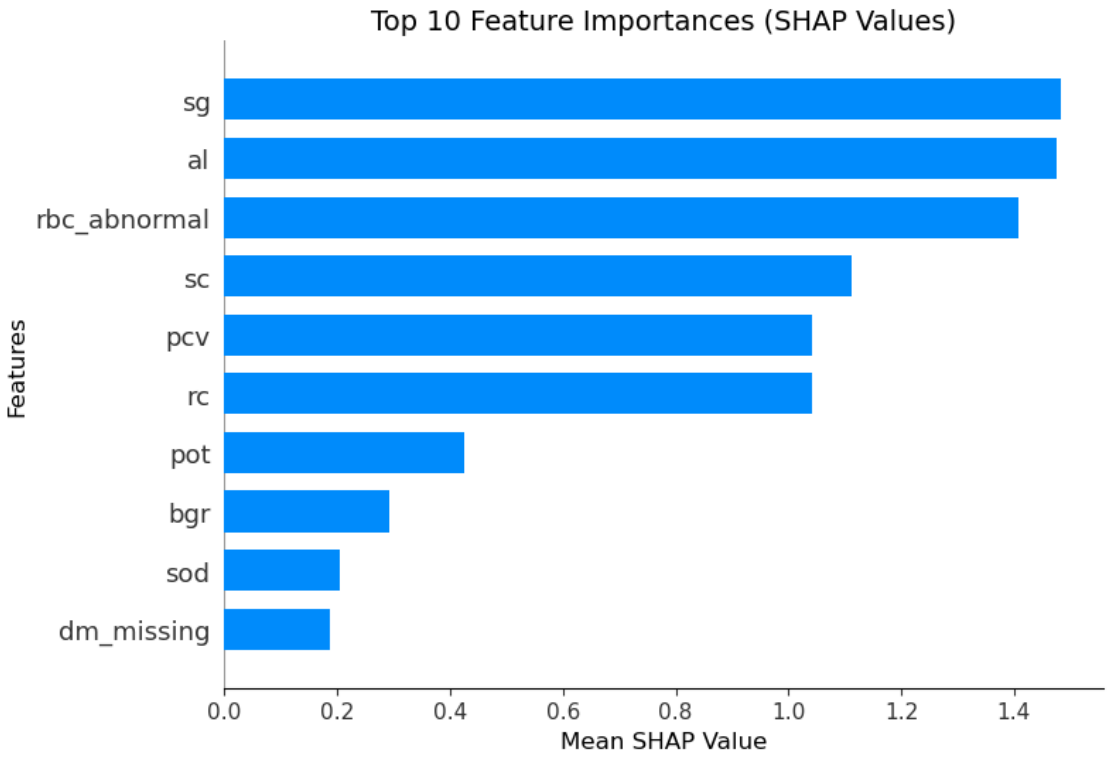

Permutation Importances (test set)
rc
al
sg
rbc_abnormal
sc
pcv
htn_no
htn_yes
htn_missing
dm_no
dm_yes
dm_missing
age
ba_missing
cad_yes
cad_missing
pe_no
pe_yes
ane_no
ane_yes
cad_no
ba_present
pcc_notpresent
pcc_missing
bp
bgr
bu
sod
pot
wc
ba_notpresent
su
rbc_normal
pc_missing
pc_normal
pc_abnormal
appet_good
pcc_present
rbc_missing
appet_poor
test score
0.95
0.96
0.97
0.98
0.99
1.00
score with perturbed feature

method2:

[51]:
```
shap.initjs()
explainer = shap.TreeExplainer(best_model_xgb[best_index])
shap_values = explainer.shap_values(X_test[best_index])
print(f'The shape of the shap values is {np.shape(shap_values)}.')
print(" ")
plt.figure(figsize=(10, 6))  # Set the figure size
shap.summary_plot(shap_values, X_test[best_index], plot_type="bar",␣
 ↪max_display=10, show=False)  # Suppress auto-show
plt.title("Top 10 Feature Importances (SHAP Values)", fontsize=14)
plt.xlabel("Mean SHAP Value", fontsize=12)
plt.ylabel("Features", fontsize=12)
plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/
 ↪shap_values_global.png', dpi = 350)
plt.show()
```

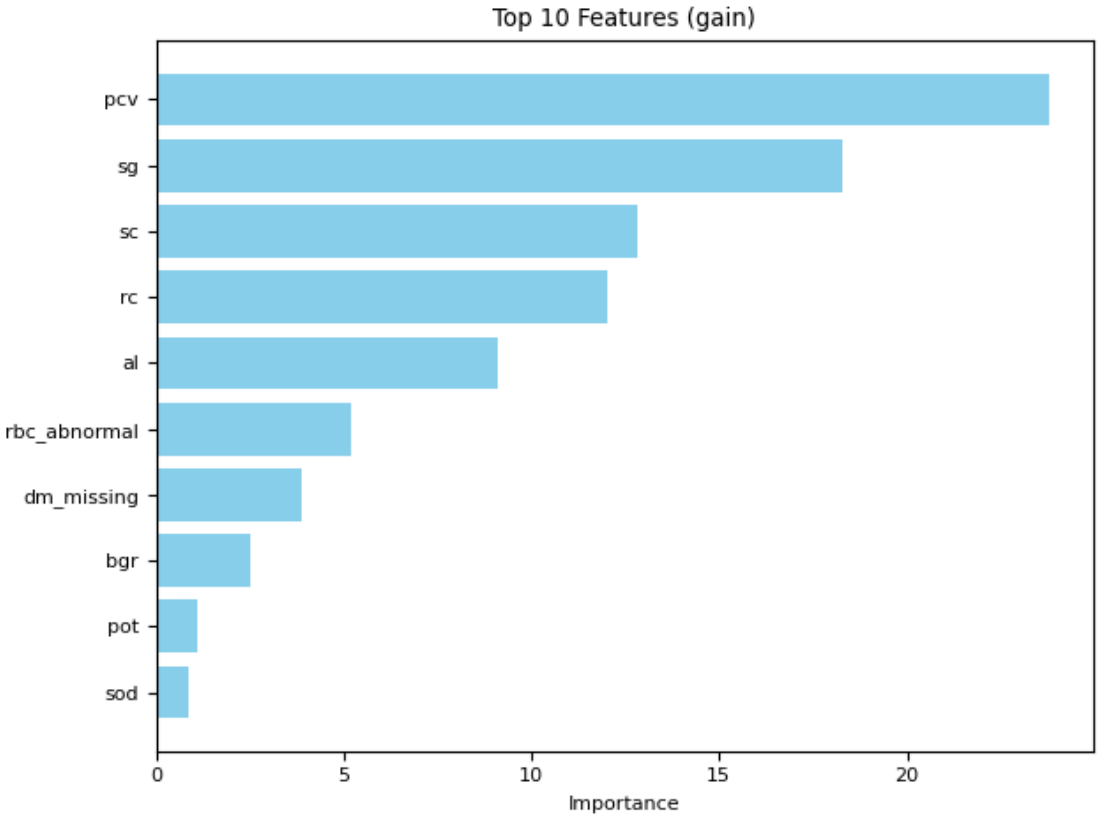
```
<IPython.core.display.HTML object>
```

```
The shape of the shap values is (80, 40).
```



method3:

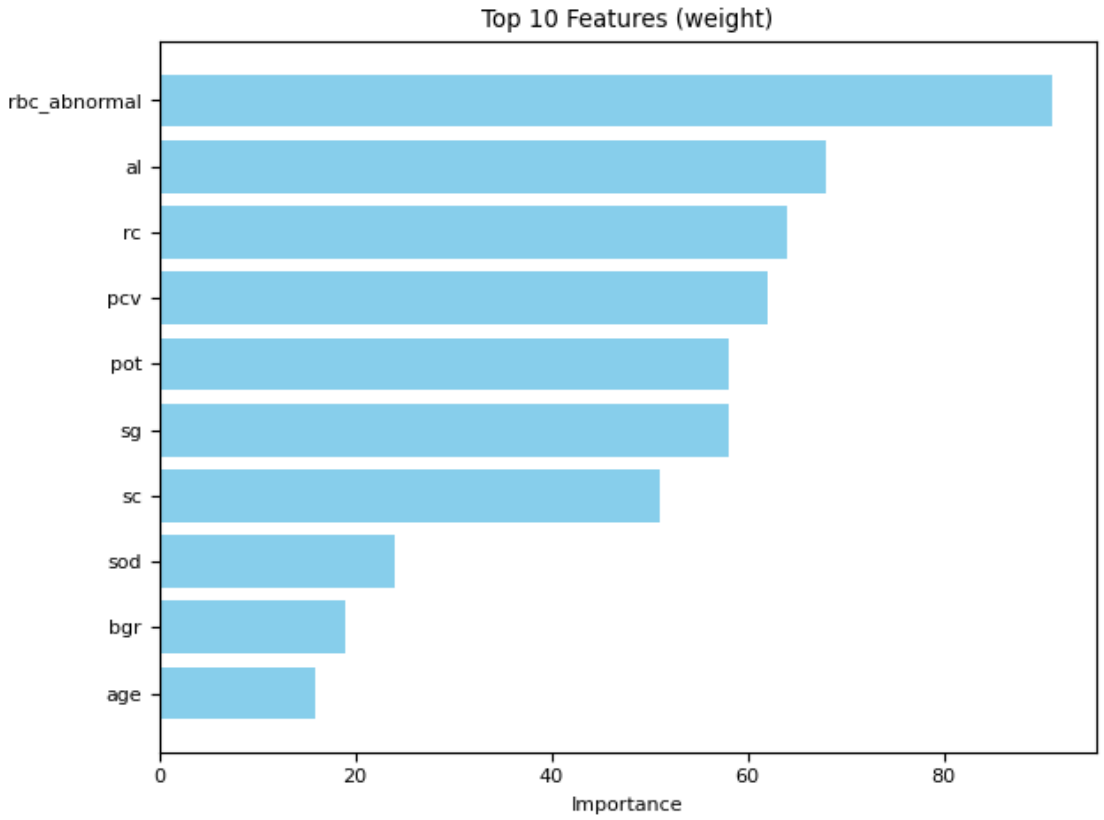
[52]:
```
booster = best_model_xgb[best_index].get_booster()
feature_importance = booster.get_score(importance_type='gain')
sorted_importance = sorted(feature_importance.items(), key=lambda x: x[1],␣
 ↪reverse=True)
feature_df = pd.DataFrame(sorted_importance, columns=['Feature', 'Importance'])
top_10_features = feature_df.head(10)
plt.barh(top_10_features['Feature'], top_10_features['Importance'],␣
 ↪color='skyblue')
plt.xlabel('Importance')
plt.title('Top 10 Features (gain)')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/imp_gain.png',␣
 ↪dpi = 350)
plt.show()
```



method 4:

[53]:
```
booster = best_model_xgb[best_index].get_booster()
feature_importance = booster.get_score(importance_type='weight')
```

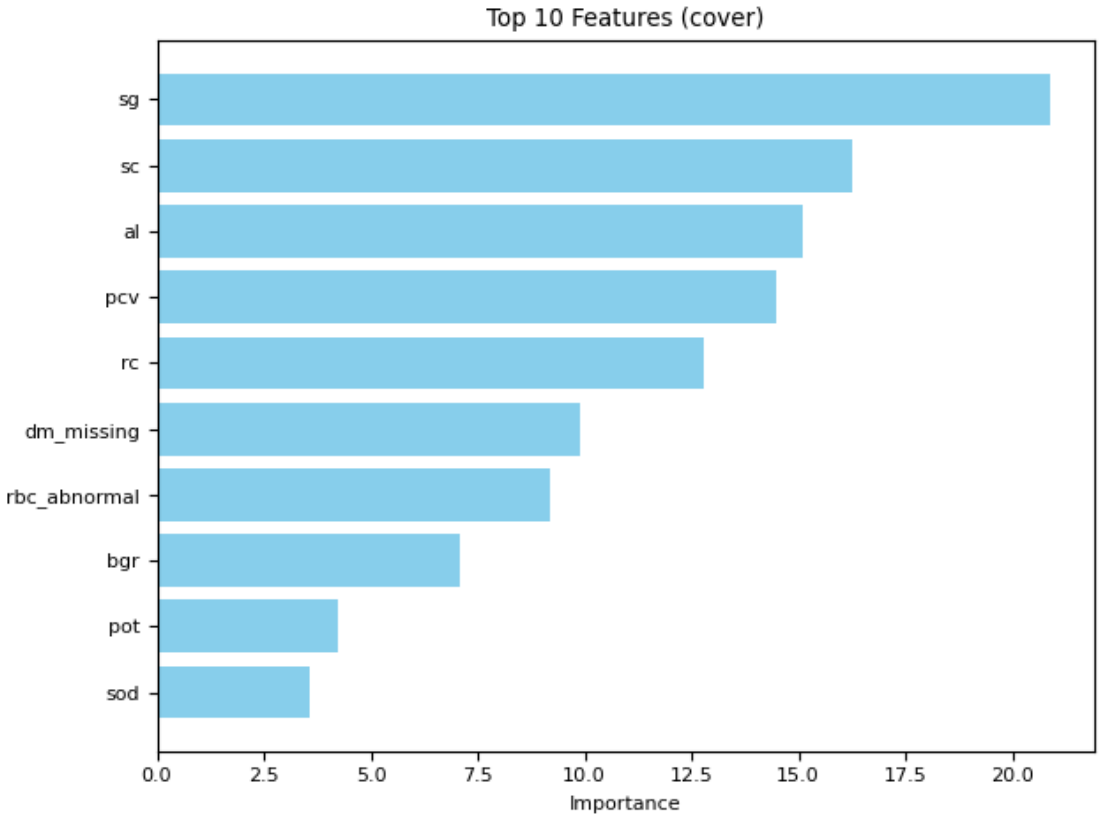
```
sorted_importance = sorted(feature_importance.items(), key=lambda x: x[1],␣
 ↪reverse=True)
feature_df = pd.DataFrame(sorted_importance, columns=['Feature', 'Importance'])
top_10_features = feature_df.head(10)
plt.barh(top_10_features['Feature'], top_10_features['Importance'],␣
 ↪color='skyblue')
plt.xlabel('Importance')
plt.title('Top 10 Features (weight)')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/imp_weight.
 ↪png', dpi = 350)
plt.show()
```



method 5:

[54]:
```
booster = best_model_xgb[best_index].get_booster()
feature_importance = booster.get_score(importance_type='cover')
sorted_importance = sorted(feature_importance.items(), key=lambda x: x[1],␣
 ↪reverse=True)
feature_df = pd.DataFrame(sorted_importance, columns=['Feature', 'Importance'])
```

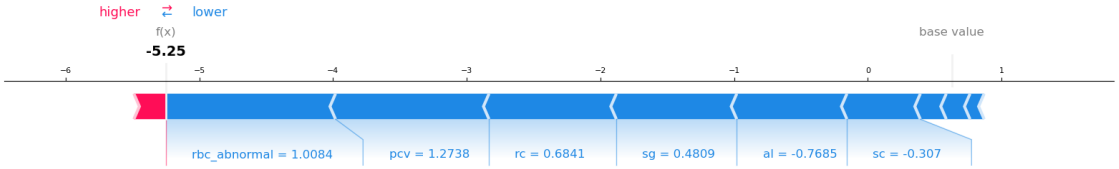
```
top_10_features = feature_df.head(10)
plt.barh(top_10_features['Feature'], top_10_features['Importance'],␣
 ↪color='skyblue')
plt.xlabel('Importance')
plt.title('Top 10 Features (cover)')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.savefig('/Users/ericdavis/Desktop/data1030/FinalProject1030/imp_cover.png',␣
 ↪dpi = 350)
plt.show()
```



local importance

[55]:
```
index = 68 # predict class 0

# Extract the SHAP values and feature values for the specific index
shap_vals = np.round(shap_values[index, :],4)
features = np.round(X_test[best_index].iloc[index, :],4)  # Feature values for␣
 ↪the specific data point

# Sort SHAP values and corresponding features by absolute SHAP value
```

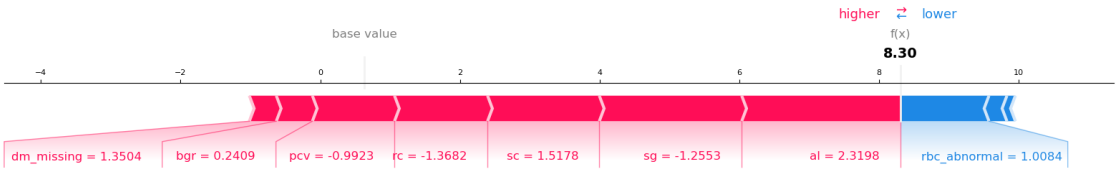
```
sorted_indices = np.argsort(np.abs(shap_vals))[-10:]  # Get indices of top 10␣
 ↪SHAP values
top_shap_vals = shap_vals[sorted_indices]
top_features = features.iloc[sorted_indices]
top_feature_names = features.index[sorted_indices]

# Plot with top 5 SHAP values
shap.force_plot(
    explainer.expected_value,
    top_shap_vals,
    features=top_features,
    feature_names=top_feature_names,
    matplotlib = True
)
plt.tight_layout()
plt.show()
```




```
<Figure size 640x480 with 0 Axes>
```

[56]:
```
index = 17 # predict class 1

# Extract the SHAP values and feature values for the specific index
shap_vals = np.round(shap_values[index, :], 4)
features = np.round(X_test[best_index].iloc[index, :], 4)  # Feature values for␣
 ↪the specific data point

# Sort SHAP values and corresponding features by absolute SHAP value
sorted_indices = np.argsort(np.abs(shap_vals))[-10:]  # Get indices of top 10␣
 ↪SHAP values
top_shap_vals = shap_vals[sorted_indices]
top_features = features.iloc[sorted_indices]
top_feature_names = features.index[sorted_indices]

# Plot with top 5 SHAP values
shap.force_plot(
    explainer.expected_value,
    top_shap_vals,
```

```
    features=top_features,
    feature_names=top_feature_names,
    matplotlib = True
)
plt.tight_layout()
plt.show()
```




```
<Figure size 640x480 with 0 Axes>
```

[ ]: