About the project

The end goal of this project is to classify patients with high protein concentration in urine and the healthy group based on SERS (Surface Enchanced Raman Spectroscopy) spectral data and biomedical data.
This project is to be released as a research paper later in 2022 or 2023. Some information might not be fully shown here as a result.

The project is divided into several Jupyter notebooks with the following names: 1) Import raw urine spectra (part 1) 2) Spectra processing (part 2) 3) Classification of patients (part 3) 4) Biomedical data (part 4) 5) Comparison of nanoparticles (part 5)

Author of all codes: Sultan Aitekenov, sultanaitekenov@gmail.com

Part of the upcoming abstract: Excessive protein excretion in human urine is an early and sensitive marker of diabetic nephropathy, primary and secondary renal disease. Kidney problems, particularly chronic kidney disease, remain among the few growing causes of mortality in the world. Therefore, it is important to develop efficient, expressive, and low-cost method for protein determination. Surface enhanced Raman spectroscopy (SERS) methods are potential candidates to achieve those criteria. In this paper, the SERS methods was developed to distinguish patients with proteinuria and the healthy group. Commercial gold nanoparticles with the diameter of 60 nm and 100 nm, and silver nanoparticles with the diameter of 100 nm were employed. Silver, gold, silicon and test slides covered with aluminium tape were utilized as substrates. Obtained spectra were analysed with several machine learning algorithms coupled with the PCA, ROC curve, and cross-validation methods.

Classification of patients (part 3)

Import data

Import modules

In [ ]:
```
# other modules related to classification are imported later
import pandas as pd
import numpy as np
import copy
import pickle
import matplotlib.pyplot as plt
```

Raman Shift

In [ ]:
```
# Si_60nm_AuNPs is not included though
```

```
raman_shift_400_1800=np.array(pd.read_csv('raman_shift_400_1800.csv', header=None))
wave = raman_shift_400_1800[0]
```

Processed urine spectra

In [ ]:
```
# data contains a nested dictionary
f = open('processed_urine_spectra.pkl', 'rb')
processed_urine_spectra = pickle.load(f)
```

In [ ]:
```
# delete 'Au_40nm_AuNPs' as it was not used for classification, see part 5
del processed_urine_spectra['Au_40nm_AuNPs']
```

Protein data and health status

Both protein data and health status will be used for labelling. They do not coincide with each other. To get more information about biomedical data see Part 5.

Protein data

In [ ]:
```
# protein data
bio_data = pd.read_csv('urine biomedicals processed.csv')
bio_data.head(10)
```

In [ ]:
```
# convert info_protein to dictionary with keys being equal to patients' ID for all pat
info_protein={}
for i in range(0,len(bio_data)):
    key_ID = bio_data['patient_ID'][i]
    info_protein[key_ID] = float(bio_data['Protein, mg/L'][i]) #converts to mg/L
```

Health status

In [ ]:
```
bio_data['status'].unique()
```

In [ ]:
```
(bio_data['status'] == 'diseased').sum()
```

In [ ]:
```
(bio_data['status'] == 'healthy').sum()
```

In [ ]:
```
bio_data['status'].count()
```

In [ ]:
```
# health status
health_status={}
for i in range(0,len(bio_data)):
    key_ID = bio_data['patient_ID'][i]
    if bio_data['status'][i] == 'diseased':
        health_status[key_ID] = 1
    elif bio_data['status'][i] == 'healthy':
        health_status[key_ID] = 0
```

Assign patients to high or low protein group

Define function that assigns to high and low protein concentration groups

In [ ]:
```
# function for creation of groups, the same structure as spectra data
def raman_protein_info(input_spectra, input_protein, protein_threshold):
    """
    Compares samples with database. Returns error if match is not found for
    samples. Reports which samples were not included.
    low protein == 0
    high protein == 1
    protein threshold in mg/L

    Output file has the same structure as input_spectra
    """
    import copy

    # copies the same structure as input spectra
    output_group = copy.deepcopy(input_spectra)

    # cycle through expiremntal sets
    for key_set in input_spectra.keys():
        # cycles through patients with key_ID
        for key_ID in input_spectra[key_set].keys():
            # take protein gramm from input_protein
            protein_gramm = input_protein.get(key_ID)
            if protein_gramm == None:
                print(f"error - your protein gramm value is not found in the database
                # delete patients who is not in the protein database
                output_group[key_set][key_ID] = None
                # del input_spectra[key_set][key_ID]
            elif protein_gramm >= protein_threshold:
                output_group[key_set][key_ID] = 1
            elif protein_gramm <= protein_threshold:
                output_group[key_set][key_ID] = 0


    output_group_final = output_group
    return output_group_final
```

For each experimental set assign to protein groups

In [ ]:
```
# protein is set to 300 mg/L as it is important threshold for medical diagnostics
protein_threshold =  300
groups_protein = raman_protein_info(processed_urine_spectra, info_protein, protein_thr
groups_protein;
```

Define function that assigns according to the health status

In [ ]:
```
def raman_health_info(input_spectra, health_status):
    """
    Compares samples with database according to their health status.
    diseased == 1
    healthy == 0

    Output file has the same structure as input_spectra
    """
```

```
    import copy

    # copies the same structure as input spectra
    output_group = copy.deepcopy(input_spectra)

    # cycle through expiremntal sets
    for key_set in input_spectra.keys():
        # cycles through patients with key_ID
        for key_ID in input_spectra[key_set].keys():

            output_group[key_set][key_ID] = health_status.get(key_ID)

    return output_group
```

For each experimental set assign to health status

In [ ]:
```
groups_health = raman_health_info(processed_urine_spectra, health_status)
```

Prepare data for x and y training sets

Select a limited range in raman shift, as it might help to improve classification scores. Uncomment or change status of the cells to Code.

select range for Raman Shift wave_left = 1355-10 wave_right = 1355+10 # create boolean array and apply it to wave select_range = (wave_left <= wave) & (wave <= wave_right) wave = wave[select_range] # output print(len(select_range)) wave# loop over experimental sets except for the set Si_60_AuNPs as it contains different data for key_set in processed_urine_spectra.keys(): if key_set != 'Si_60nm_AuNPs': #loop over patients' IDs for key_ID in processed_urine_spectra[key_set].keys(): spectrum = processed_urine_spectra[key_set][key_ID] limited_spectrum = spectrum[select_range] processed_urine_spectra[key_set][key_ID] = limited_spectrum

Convert dictionaries to matrices

In [ ]:
```
# create empty dict
x = {}
y = {}
IDs = {}


for key_set in processed_urine_spectra.keys():
        # create empty matrix to assign to them values later
        matrix_x = []
        matrix_y = []
        matrix_IDs = []

        # loop to make dict into matrix
        for key_ID in processed_urine_spectra[key_set].keys():
            matrix_x.append( processed_urine_spectra[key_set][key_ID] )
            matrix_y.append( groups_health[key_set][key_ID] )
            # matrix_y.append( groups_protein[key_set][key_ID] )
            matrix_IDs.append( int(key_ID) )

        # assign matrix to x values
        x[key_set] = np.array(matrix_x)
```

```
        y[key_set] = np.array(matrix_y)
        IDs[key_set] = np.array(matrix_IDs)
```

Spectra high vs low protein groups

In [ ]:
```
# plot spectra of high vs low protein groups for every substrate

for key_set in x.keys():
    plt.figure(figsize =(30,10))
    for i in range(0, len(x[key_set])):
        # low protein
        if y[key_set][i] == 0 and key_set != 'Si_60nm_AuNPs':
            plt.subplot(1,2,1)
            plt.plot(wave, x[key_set][i])
            plt.xlabel('Raman shift, cm-1')
            plt.ylabel('Raman intensity, a.u.')
            plt.title(f'{key_set} - low protein')
        # high protein
        elif y[key_set][i] == 1 and key_set != 'Si_60nm_AuNPs':
            plt.subplot(1,2,2)
            plt.plot(wave, x[key_set][i])
            plt.xlabel('Raman shift, cm-1')
            plt.ylabel('Raman intensity, a.u.')
            plt.title(f'{key_set} - high protein')
```

Average spectra

In [ ]:
```
# plot average spectra of high vs low protein groups for every substrate
x_low = copy.deepcopy(x)
x_high = copy.deepcopy(x)

# create variables that contain mean values
for key_set in x.keys():
    # purge data from x_low and x_high
    x_low[key_set] = np.zeros((2045))
    x_high[key_set] = np.zeros((2045))
    # cycle through patients to categorize in two groups
    for i in range(0, len(x[key_set])):
        # low protein
        if y[key_set][i] == 0 and key_set != 'Si_60nm_AuNPs':
            x_low[key_set] = np.vstack((x_low[key_set], x[key_set][i]))
        # high protein
        elif y[key_set][i] == 1 and key_set != 'Si_60nm_AuNPs':
            x_high[key_set] = np.vstack((x_high[key_set], x[key_set][i]))


# create mean values
for key_set in x_low.keys():
    # delete row of zeros
    x_low[key_set] = np.delete(x_low[key_set], 0, axis = 0)
    x_high[key_set] = np.delete(x_high[key_set], 0, axis = 0)
    # create mean afterwards
    x_low[key_set] = np.mean(x_low[key_set], axis = 0)
    x_high[key_set] = np.mean(x_high[key_set], axis = 0)

# plots
```

```
for key_set in x_low.keys():
    plt.figure(figsize =(20,10))
    if key_set != 'Si_60nm_AuNPs':
        plt.plot(wave, x_low[key_set], label = 'low protein')
        plt.plot(wave, x_high[key_set], label = 'high protein')
        plt.plot(wave, x_high[key_set]-x_low[key_set], label = 'high - low')
        plt.xlabel('Raman shift, cm-1')
        plt.ylabel('Raman intensity, a.u.')
        plt.title(f'{key_set}')
        plt.legend()
```

Classification part

PCA

PCA 99%

In [ ]:
```
# PCA
from sklearn.decomposition import PCA

# perform PCA analysis on all set
pca = PCA(0.999)
x_pca = copy.deepcopy(x)
for key_set in x.keys():
    x_pca[key_set] = pca.fit(x[key_set]).transform(x[key_set]) # xx_pca=pca.fit_transf
    print(f'{key_set}' + f' - PCA components {pca.n_components_}\n')
```

PCA 2 components

In [ ]:
```
# perform PCA analysis on all set
pca = PCA(n_components = 15)
x_pca_2 = copy.deepcopy(x)
for key_set in x.keys():
    x_pca_2[key_set] = pca.fit(x[key_set]).transform(x[key_set]) # xx_pca=pca.fit_trar
    print(f'{key_set}' + f' - PCA components {pca.n_components_}\n')
```

Estimators preparation

In [ ]:
```
# Estimators:
# LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
# logistic regression
from sklearn.linear_model import LogisticRegression
# KNN
from sklearn.neighbors import KNeighborsClassifier
# SVM
from sklearn.svm import SVC
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
# RFC
from sklearn.ensemble import RandomForestClassifier
```

```
# Naive Gaussian
from sklearn.naive_bayes import GaussianNB

# create dictionary of models
models_dict = {'LogisticRegression': LogisticRegression(solver =  'liblinear'),
               'Linear Discriminant Analysis': LinearDiscriminantAnalysis(),
                'Gaussian Naive Bayes': GaussianNB(),
               'K-nearest Neighbors': KNeighborsClassifier(10),
               #'Support Vector Machnines': SVC(),
               #'Decision Tree': DecisionTreeClassifier(),
               'Random Forest': RandomForestClassifier()}

# create dict of scores that will be overwritten
empty_models_dict = copy.deepcopy(models_dict)
for key_model in empty_models_dict.keys():
    empty_models_dict[key_model] = []

# scores dict need to have key_set
scores_dict = {}
scores_dict_cv = {}
scores_dict_matlab = {}
probability_dict = {}
probability_dict_cv = {}
auc_values = {}
auc_values_cv = {}
for key_set in x.keys():
    scores_dict[key_set] = copy.deepcopy(empty_models_dict)
    scores_dict_cv[key_set] = copy.deepcopy(empty_models_dict)
    probability_dict[key_set] = copy.deepcopy(empty_models_dict)
    probability_dict_cv[key_set] = copy.deepcopy(empty_models_dict)
    auc_values[key_set] = copy.deepcopy(empty_models_dict)
    auc_values_cv[key_set] = copy.deepcopy(empty_models_dict)

# see how it looks
scores_dict;
```

In [ ]:
```
def get_score(model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    return round(model.score(X_test, y_test),2)

from sklearn.metrics import roc_curve, roc_auc_score
def get_probability(model, X_train, X_test, y_train, y_test):
    model_fitted = model.fit(X_train, y_train)
    probability_scores = model_fitted.predict_proba(X_test)
    # probabilities for positive outcomes
    probability_scores = probability_scores[:,1]
    return probability_scores
```

Training and testing sets are the same

PCA 2 components data from Python

In [ ]:
```
# train model and calculate accuracy

X_train = x_pca_2
X_test = x_pca_2
```

```
y_train = y
y_test = y
for key_set in X_train.keys():
    for key_model in models_dict.keys():
        scores_dict[key_set][key_model] = get_score(models_dict[key_model],
                                                    X_train[key_set], X_test[key_set], y_train[

# present data in the intuitive format for accuracy
df_scores_dict_2 = pd.DataFrame.from_dict(scores_dict)
df_scores_dict_2
```

2D plots of PCA

In [ ]:
```
import sklearn
print('The scikit-learn version is {}.'.format(sklearn.__version__))
```

In [ ]:
```
# import matplotlib.pyplot as plt
# from sklearn.linear_model import LogisticRegression
# from sklearn.inspection import DecisionBoundaryDisplay

# key_set = 'Ag_100nm_AuNPs'

# classifier = LogisticRegression().fit(x_pca_2[key_set], y[key_set])
# disp = DecisionBoundaryDisplay.from_estimator(
#     classifier, X, response_method="predict",
#     # xlabel=iris.feature_names[0], ylabel=iris.feature_names[1],
#     alpha=0.5,
# )
# disp.ax_.scatter(X[:, 0], X[:, 1], edgecolor="k")

# plt.show()
```

PCA 99%

In [ ]:
```
# train model and calculate accuracy
X_train = x_pca
X_test = x_pca
y_train = y
y_test = y
for key_set in X_train.keys():
    for key_model in models_dict.keys():
        scores_dict[key_set][key_model] = get_score(models_dict[key_model],
                                                    X_train[key_set], X_test[key_set], y_train[
# present data in the intuitive format for accuracy
df_scores_dict = pd.DataFrame.from_dict(scores_dict)
df_scores_dict
```

In [ ]:
```
# for key_set in scores_dict.keys():
#     scores_dict[key_set]['average for set'] = []
#     for key_model in scores_dict[key_set].keys():
#         scores_dict[key_set]['average for set'].append(scores_dict[key_set][key_mode
#         print(scores_dict[key_set][key_model])
#         print(scores_dict[key_set]['average for set'])
#     print(key_set)
#     print(scores_dict[key_set]['average for set'])
#     scores_dict[key_set]['average for set'] = np.mean(np.array(scores_dict[key_set][
```

ROC curves

In [ ]:
```
from sklearn.metrics import roc_curve, roc_auc_score

for key_set in X_train.keys():
    for key_model in models_dict.keys():
        probability_dict[key_set][key_model] = get_probability(models_dict[key_model],
                                                X_train[key_set], X_test[key_set], y_tr


for key_set in y_test.keys():
    for key_model in probability_dict[key_set].keys():
        auc_values[key_set][key_model] = roc_auc_score(y_test[key_set], probability_di

df_auc_values = pd.DataFrame.from_dict(auc_values)

df_auc_values
```

Crossvalidation

Stratified K fold crossvalidation for data from Python, manually written

In [ ]:
```
# Create stratified crossvalidation with K-Folds
K = 10 # number of folds
from sklearn.model_selection import StratifiedKFold
folds = StratifiedKFold(n_splits = K)

# train model for crossvalidation
for key_set in x_pca.keys():
    for train_index, test_index in folds.split(x_pca[key_set],y[key_set]):
        X_train, X_test, y_train, y_test = x_pca[key_set][train_index], x_pca[key_set]
        for key_model in models_dict.keys():
            scores_dict_cv[key_set][key_model].append(get_score(models_dict[key_model]
            auc_values_cv[key_set][key_model].append(roc_auc_score(y_test, get_probabi
```

In [ ]:
```
# create mean values for scores and AUC from K fold crossvalidation
for key_set in scores_dict_cv.keys():
    #scores_dict_cv[key_set]['average'] = 0
    #auc_values_cv[key_set]['average'] = 0
    for key_model in scores_dict_cv[key_set].keys():
        scores_dict_cv[key_set][key_model] = [round(np.mean(scores_dict_cv[key_set][ke
        auc_values_cv[key_set][key_model] = [round(np.mean(auc_values_cv[key_set][key_
        #scores_dict_cv[key_set]['average'] = np.vstack(round(np.mean(scores_dict_cv[k
        #auc_values_cv[key_set]['average'] = np.vstack(round(np.mean(auc_values_cv[key

# for key_set in scores_dict_cv.keys():
#     scores_dict_cv[key_set]['average'] = np.mean(scores_dict_cv[key_set]['average'])
#     auc_values_cv[key_set]['average'] = np.mean(auc_values_cv[key_set]['average'])
```

In [ ]:
```
# table of scores and standart deviation
df_scores_dict_cv = pd.DataFrame.from_dict(scores_dict_cv)
df_scores_dict_cv
```

In [ ]:
```
# table of auc values and standart deviation
df_auc_values_cv = pd.DataFrame.from_dict(auc_values_cv)
df_auc_values_cv
```

In [ ]:
```
# save files to excel
with pd.ExcelWriter("auc_values.xlsx") as writer:
    df_auc_values.to_excel(writer, sheet_name="auc no crossval")
    df_auc_values_cv.to_excel(writer, sheet_name="auc 10 crossval")
```

Learning curve

In [ ]:
```
# Learning curve for Ag_100nm_AuNPs as it has the best performance
from sklearn.model_selection import learning_curve

# plot learning curve

key_set = 'Ag_100nm_AuNPs'
train_sizes, train_scores, valid_scores = learning_curve(KNeighborsClassifier(), x_pca
train_scores_mean = train_scores.mean(axis = 1)
valid_scores_mean = valid_scores.mean(axis = 1)
plt.figure(figsize = [15,10])
plt.plot(train_sizes, train_scores_mean, "o--", label = 'training scores')
plt.plot(train_sizes, valid_scores_mean, "o--", label = 'validation scores')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('training size')
```

In [ ]:
```
# Learning curve for Al_tape_60nm_AuNPs as it has the worst performance
from sklearn.model_selection import learning_curve

# plot learning curve

key_set = 'Al_tape_60nm_AuNPs'
train_sizes, train_scores, valid_scores = learning_curve(KNeighborsClassifier(), x_pca
train_scores_mean = train_scores.mean(axis = 1)
valid_scores_mean = valid_scores.mean(axis = 1)
plt.figure(figsize = [15,10])
plt.plot(train_sizes, train_scores_mean, "o--", label = 'training scores')
plt.plot(train_sizes, valid_scores_mean, "o--", label = 'validation scores')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('training size')
```

cross_val_score for data from Python

In [ ]:
```
# cross_val_score for data from Python and MATLAB
# a and b in the loop are just for easy naming
from sklearn.model_selection import cross_val_score
for key_model in models_dict.keys():
    # Au_100nm_AuNPs
    a_python = cross_val_score(models_dict[key_model], x_pca_2['Au_100nm_AuNPs'], y['A
    # Ag_100nm_AgNPs
    b_python = cross_val_score(models_dict[key_model], x_pca_2['Ag_100nm_AgNPs'], y['A
    # print scores
```

```
    print(f'A) accuracy for Au_100nm_AuNPs python {round(a_python,2)}')
    print(f'B) accuracy for Ag_100nm_AgNPs python {round(b_python,2)}')
```

Optimize KNN clasifier

In [ ]:

```
# optimize KNN by changing number of neighbors
key_set = 'Ag_100nm_AuNPs'

for i in range(1,15):
    auc_values_KNN_cv = round(cross_val_score(KNeighborsClassifier(i), x_pca[key_set],
    acc_values_KNN_cv = round(cross_val_score(KNeighborsClassifier(i), x_pca[key_set],

    print(f'KNN accuracy for {i} neighbors and accuracy value {acc_values_KNN_cv} and
```