baseline_model_report

March 14, 2020

ChBE 6745 Project: Prediction of Adsorption Properties of Metal-Organic Frameworks with Framework Flexibility

Chao-Wen Chang, Pengfei Cheng, Po-Wei Huang, Xiaohan Yu, Yamin Zhang

0.1 Problem definition

Agrawal and Sholl (2019) show that appropriate consideration of framework flexibility may be important to quantitative predictions about molecular adsorption in metal-organic frameworks (MOFs). However, taking the framework flexibility into account directly in the molecular simulation framework may be 5-10 times morecomputationally expensive than standard simulation methods based on rigid crystal structure. Therefore, we are interested in constructing a machine learning model to predict the adsorption properties of MOFs with framework flexibility based on: (1) the features of the MOFs, (2) the features of adsorbates, and (3) the adsorption properties from standard simulation methods based on rigid crystal structure.

0.2 Dataset details

The dataset is in the `data` folder. It is composed of three parts: 1. **29 MOF features** (under `data/ML_data`) 2. **6 adsorbate features** (manually added below) 3. **adsorption uptakes** of **801 (MOF, adsorbate) pairs** (under `data/flexibility_data/y_data/adsorption_data`), containing two values: 1. values from rigid model 2. mean values from flexible model

0.3 Model training strategy

1. Multi-linear regression
2. Kernel regression
3. Kernel ridge regression
4. Lasso regression
5. Genetic programming
6. Neural network

0.4 Model validation strategy

1. hold-out
2. k-fold
3. bootstrapping

0.5 Reference

Agrawal, Mayank, and David S. Sholl. "Effects of Intrinsic Flexibility on Adsorption Properties of Metal–Organic Frameworks at Dilute and Nondilute Loadings." *ACS applied materials & interfaces* 11.34 (2019): 31060-31068.

1 Data preprocessing

This section contains the codes for preprocessing the datasets to generate two numpy arrays in Python ("X" and "y"), dropping non-numerical rows, and feature scaling.

1.1 Read MOF features

```
In [1]: import pandas as pd
        import numpy as np
        import os
        import matplotlib.pyplot as plt
        %matplotlib inline

        # read the 36-descriptor data
        df36Descriptor = pd.read_excel('data/ML_data/descriptor_used.xlsx',header=4,index_col=

        # clean up the column
        columns = [df36Descriptor.columns[1]] + df36Descriptor.columns[3: -11].tolist()

        newColumns = {}
        for ci in columns:
            if ' ' in ci:
                newColumns[ci] = ci.split(' ',1)[0]
            elif '(' in ci:
                newColumns[ci] = ci.split('(',1)[0]
            else:
                newColumns[ci] = ci

        dfShortNames = df36Descriptor[columns].rename(columns=newColumns)

        # reduce columns to only contain MOF features
        shared_descriptor = [col for col in dfShortNames.columns if col in newColumns]
        dfMLReduced = dfShortNames[shared_descriptor]

        dfMLReduced.head()
```

```
Out[1]:                MOF        vf  nAT-H       nNM        nM       nTB       nSB  \
        Isotherm ID
        1           ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476
        1           ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476
        1           ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476
        1           ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476
```

```
2             ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476

                   nMB       nRB       nR6    ...        nR4  MType  MaxMVal  \
Isotherm ID                                   ...
1             0.380952  0.952381  0.166667    ...    0.02381      1        4
1             0.380952  0.952381  0.166667    ...    0.02381      1        4
1             0.380952  0.952381  0.166667    ...    0.02381      1        4
1             0.380952  0.952381  0.166667    ...    0.02381      1        4
2             0.380952  0.952381  0.166667    ...    0.02381      1        4

                  n-O-  F01[H-C]  F01[C-N]  F01[C-O]  F02[H-C]  F02[C-N]  \
Isotherm ID
1             0.238095  0.285714       0.0  0.285714  0.571429       0.0
1             0.238095  0.285714       0.0  0.285714  0.571429       0.0
1             0.238095  0.285714       0.0  0.285714  0.571429       0.0
1             0.238095  0.285714       0.0  0.285714  0.571429       0.0
2             0.238095  0.285714       0.0  0.285714  0.571429       0.0

              F02[C-O]
Isotherm ID
1             0.285714
1             0.285714
1             0.285714
1             0.285714
2             0.285714

[5 rows x 29 columns]
```

1.2 Read adsorption update data

```
In [2]: # the MOFs in "dfMLReduced" and adsorption data sets are different, so it is necessary
        def datasetMatch(MOFName):
            dfML= dfMLReduced[dfMLReduced['MOF'].isin(MOFName)].drop_duplicates()
            matchedMOFIndex=np.isin(MOFName, dfML['MOF'].values)
            return matchedMOFIndex, dfML

        # read flexibility data
        flexibilityList=os.listdir('data/flexibility_data/y_data/adsorption_data') # obtain li
        flexivilityData=[]
        adsorbateNameList = []

        for i, name in enumerate(flexibilityList):
            # read csv files for certain adsorption uptakes
            df = pd.read_csv('data/flexibility_data/y_data/adsorption_data/' + name)

            # obtain the rigid value
            rigidValue = np.array(df[df.columns[1]], dtype = float)
            # obtain the flexible mean value
            flexValue = np.mean(np.array(df[df.columns[2:]],dtype=float),axis=1)

            # obtain the adsorbate label
            label = np.array([name.split("_")[1] for x in range(0,len(flexValue))],dtype=str)
            adsorbateNameList.append(name.split("_")[1])

            # stack the rigid value, flexible mean value and the adsorbate label
            singleSet = np.column_stack([rigidValue,flexValue,label])

            if i == 0:
                # obtain the name list of MOFs
                MOFNameTemp = np.array(df[df.columns[0]], dtype = str)
                MOFName = [x.split("_")[0] for x in MOFNameTemp]

                # search the MOF name in "dfMLReduced", generating dfML
                matchedMOFIndex, dfML = datasetMatch(MOFName)
                print("The number of MOFs shared by two datasets are: {:d}.".format(dfML.shape

                # generating flexibilityData as "y"
                flexibilityData = singleSet[matchedMOFIndex,:].copy()
            else:
                # concatenate "y"
                flexibilityData = np.concatenate([flexibilityData.copy(),singleSet[matchedMOFI

        print("The total number of data points are: {}.".format(flexibilityData.shape[0]))

        dfML.head()
```

```
The number of MOFs shared by two datasets are: 89.
The total number of data points are: 801.
```

```
Out[2]:                 MOF        vf  nAT-H       nNM        nM       nTB  \
        Isotherm ID
        1            ABUWOJ  0.545974    168  1.238095  0.095238  1.571428
        26           ACOLIP  0.454051    256  1.562500  0.031250  1.750000
        72           AGARUW  0.450504    224  1.071428  0.071429  1.678572
        97         AHOKIR01  0.460404    112  1.428572  0.142857  1.928572
        117          AMILUE  0.566397    176  1.500000  0.045455  1.795454

                          nSB       nMB       nRB       nR6    ...          nR4  \
        Isotherm ID                                          ...
        1            1.190476  0.380952  0.952381  0.166667    ...     0.023810
        26           1.250000  0.125000  0.687500  0.062500    ...     0.000000
        72           1.535714  0.642857  1.464286  0.357143    ...     0.214286
        97           1.928572  0.571429  1.000000  0.142857    ...     0.000000
        117          1.454546  0.204545  1.045454  0.181818    ...     0.000000
```

```
              MType  MaxMVal      n-O-  F01[H-C]  F01[C-N]  F01[C-O]  F02[H-C]  \
Isotherm ID
1                 1        4  0.238095  0.285714  0.000000  0.285714  0.571429
26                1        4  0.062500  0.531250  0.343750  0.125000  0.750000
72                1        9  0.214286  0.071429  0.142857  0.392857  0.214286
97                1        4  0.285714  0.571429  0.000000  0.000000  0.857143
117               1        5  0.170455  0.545455  0.272727  0.181818  0.545455

             F02[C-N]  F02[C-O]
Isotherm ID
1            0.000000  0.285714
26           0.250000  0.125000
72           0.428571  0.285714
97           0.000000  0.428571
117          0.000000  0.181818

[5 rows x 29 columns]
```

flexibilityData contains the adsorption update data for (MOF, adsorbate) pairs. There are 89 MOFs and 9 adsorbates, so there are 801 data points in total. - 1st column: rigid data - 2nd column: flexible mean data - 3rd column: adsorbate label

The order of the flexibilityData is:

MOF	adsorbate
MOF1	adsorbate1
MOF2	adsorbate1
MOF3	adsorbate1
...	...
MOF89	adsorbate1
MOF1	adsorbate2
MOF2	adsorbate2
MOF3	adsorbate2
...	...
MOF89	adsorbate2
MOF1	adsorbate3
MOF2	adsorbate3
MOF3	adsorbate3
...	...

1.3 Manually add adsorbate features

```
In [3]: # manually add adsorbate descriptors

        # Mw/gr.mol-1, Tc/K, Pc/bar, , Tb/K, Tf/K

        adsorbateData=np.array([
            ['xenon',131.293,289.7,58.4,0.008,164.87,161.2],
            ['butane',58.1,449.8,39.5,0.3,280.1,146.7],
            ['propene',42.1,436.9,51.7,0.2,254.8,150.6],
            ['ethane',30.1,381.8,50.3,0.2,184.0,126.2],
            ['propane',44.1,416.5,44.6,0.2,230.1,136.5],
            ['CO2',44.0,295.9,71.8,0.2,317.4,204.9],
            ['ethene',28.054,282.5,51.2,0.089,169.3,228],
            ['methane',16.04,190.4,46.0,0.011,111.5,91],
            ['krypton',83.798,209.4,55.0,0.005,119.6,115.6]])

        adsorbateData.shape
        adDf = pd.DataFrame(data=adsorbateData, columns=["adsorbate", "Mw/gr.mol-1", "Tc/K", "I

        # sort the dataframe based on adsorbateNameList
        sorterIndex = dict(zip(adsorbateNameList,range(len(adsorbateNameList))))
        adDf['an_Rank'] = adDf['adsorbate'].map(sorterIndex)
        adDf.sort_values(['an_Rank'],ascending = [True], inplace = True)
        adDf.drop('an_Rank', 1, inplace = True)
        adDfFloat = adDf.iloc[:, 1:].astype(np.float)
        adDfFloat["adsorbate"] = adDf["adsorbate"]
        adDfFloat
```

```
Out[3]:    Mw/gr.mol-1   Tc/K  Pc/bar           Tb/K   Tf/K adsorbate
        4       44.100  416.5    44.6  0.200  230.10  136.5   propane
        1       58.100  449.8    39.5  0.300  280.10  146.7    butane
        5       44.000  295.9    71.8  0.200  317.40  204.9       CO2
        8       83.798  209.4    55.0  0.005  119.60  115.6   krypton
        3       30.100  381.8    50.3  0.200  184.00  126.2    ethane
        2       42.100  436.9    51.7  0.200  254.80  150.6   propene
        0      131.293  289.7    58.4  0.008  164.87  161.2     xenon
        7       16.040  190.4    46.0  0.011  111.50   91.0   methane
        6       28.054  282.5    51.2  0.089  169.30  228.0    ethene
```

```
In [4]: print(adDfFloat.shape)
```

```
(9, 7)
```

There are 6 descriptors (excluding name label) for each adsorbate.

1.4 Combine MOF and adsorbate descriptors

The combined dataset should have $29 + 6 = 35$ descriptors:

```
In [5]: # replicate dfML for 9 adsorbates
        dfMLReplicate = pd.concat([dfML]*9)

        # replicate adDf for 89 MOFs
        adDfReplicate = pd.DataFrame(np.repeat(adDfFloat.values,89,axis=0))
        adDfReplicate.columns = adDfFloat.columns

        # concatenate two datasets
        dfMLReplicate.reset_index(drop=True, inplace=True)
        adDfReplicate.reset_index(drop=True, inplace=True)
        XAllDescriptor = pd.concat([dfMLReplicate, adDfReplicate],axis=1)
        print(XAllDescriptor.shape)
        XAllDescriptor.head()
```

```
(801, 36)
```

```
Out[5]:         MOF        vf  nAT-H       nNM        nM       nTB       nSB  \
        0    ABUWOJ  0.545974    168  1.238095  0.095238  1.571428  1.190476
        1    ACOLIP  0.454051    256  1.562500  0.031250  1.750000  1.250000
        2    AGARUW  0.450504    224  1.071428  0.071429  1.678572  1.535714
        3  AHOKIR01  0.460404    112  1.428572  0.142857  1.928572  1.928572
        4    AMILUE  0.566397    176  1.500000  0.045455  1.795454  1.454546

                nMB       nRB       nR6    ...      F02[H-C]  F02[C-N]  F02[C-O]  \
        0  0.380952  0.952381  0.166667    ...      0.571429  0.000000  0.285714
        1  0.125000  0.687500  0.062500    ...      0.750000  0.250000  0.125000
        2  0.642857  1.464286  0.357143    ...      0.214286  0.428571  0.285714
        3  0.571429  1.000000  0.142857    ...      0.857143  0.000000  0.428571
        4  0.204545  1.045454  0.181818    ...      0.545455  0.000000  0.181818

           Mw/gr.mol-1  Tc/K  Pc/bar        Tb/K   Tf/K  adsorbate
        0         44.1  416.5    44.6  0.2  230.1  136.5    propane
        1         44.1  416.5    44.6  0.2  230.1  136.5    propane
        2         44.1  416.5    44.6  0.2  230.1  136.5    propane
        3         44.1  416.5    44.6  0.2  230.1  136.5    propane
        4         44.1  416.5    44.6  0.2  230.1  136.5    propane

        [5 rows x 36 columns]
```

1.5 generate X and y

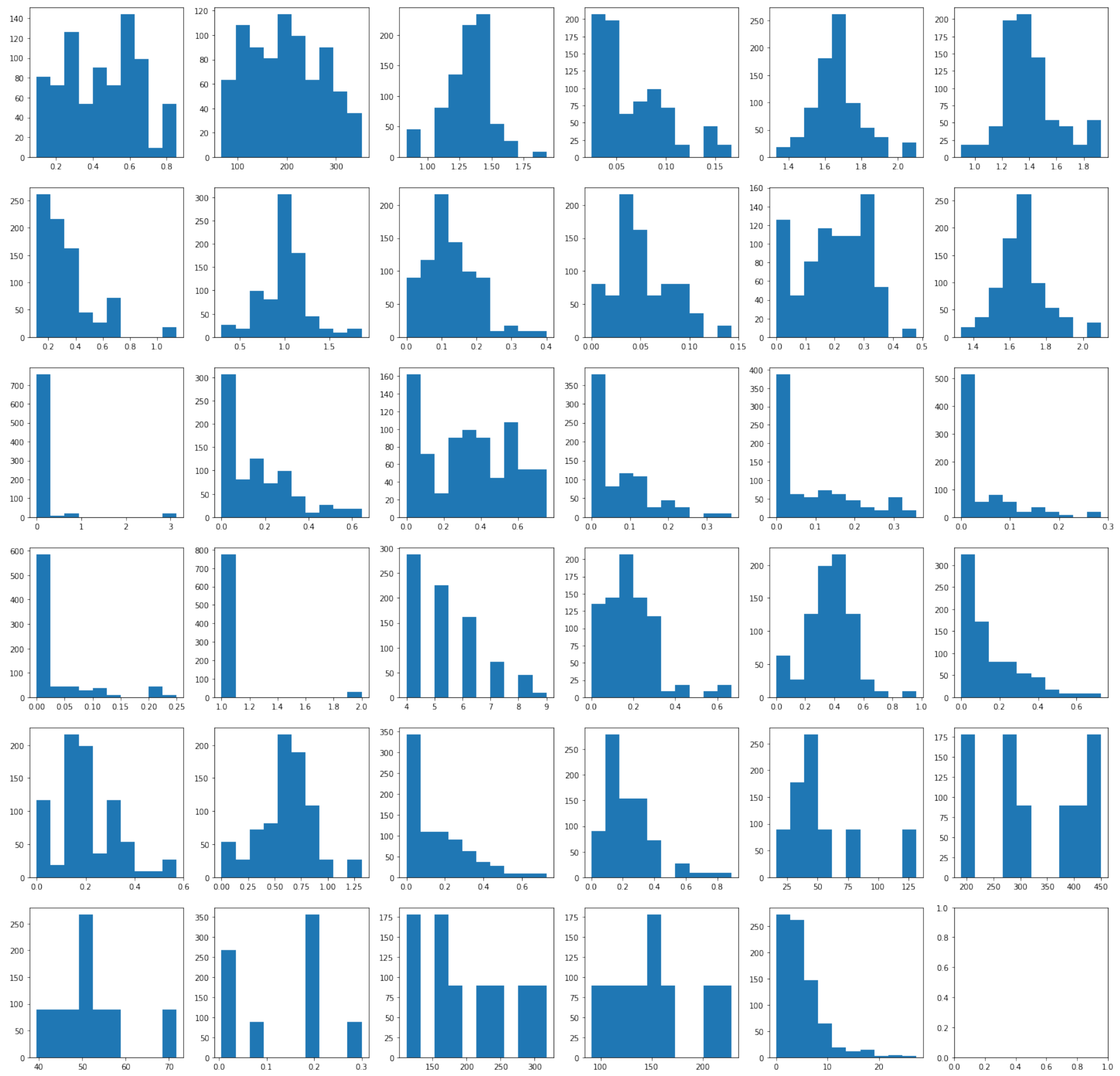
The rigid uptake data can be added into X, while the flexible mean data is chosen as y:

```
In [6]: X = np.concatenate((XAllDescriptor.iloc[:, 1:-1], flexibilityData[:, 0].reshape(-1, 1)
        y = flexibilityData[:, 1].astype('float64') .reshape(-1,1)
```

```
In [7]: # # Remove rows that contains NaN
        # X_y = np.append(X, y, axis=1)
        # print(X_y.shape)
        # print(np.isnan(X_y).any())

        # X_y = X_y[~np.isnan(X_y).any(axis=1)]
        # print(X_y.shape)
        # print(np.isnan(X_y).any())

        # X, y = X_y[:,:-2], X_y[:,-1]
        # np.linalg.matrix_rank(X)
```

1.6 Visualize distributions of features

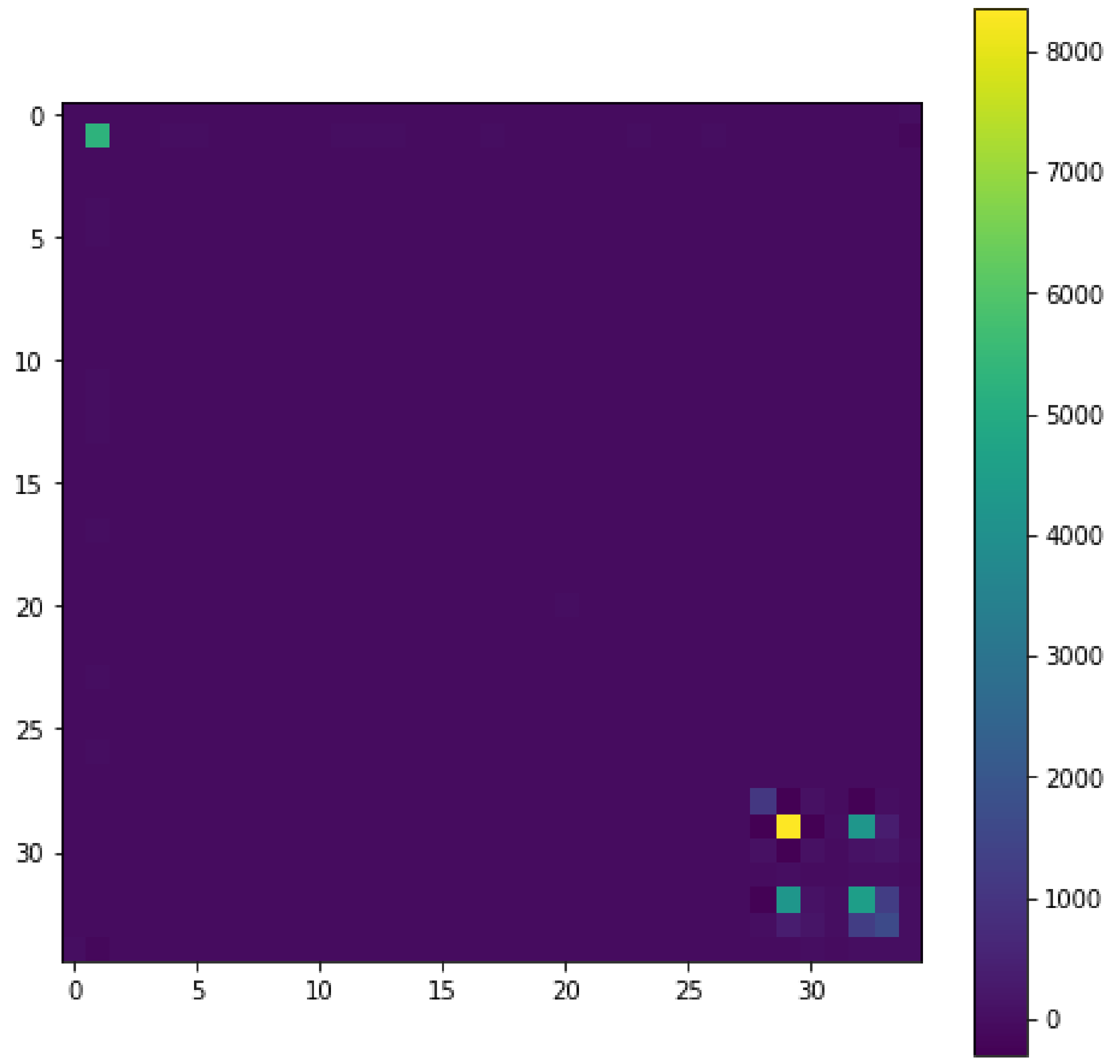
```
In [8]: N = X.shape[-1]
        n = int(np.sqrt(N))
        fig, axes = plt.subplots(n+1, n+1, figsize = (5*n, 5*n))
        ax_list = axes.ravel()
        for i in range(N):
            ax_list[i].hist(X[:,i])
```



1.7 Feature scaling

```
In [9]: covar = np.cov(X.T)

        fig,ax = plt.subplots(figsize=(8, 8))
        c = ax.imshow(covar)
        fig.colorbar(c)
```

```
Out[9]: <matplotlib.colorbar.Colorbar at 0x117ec5278>
```

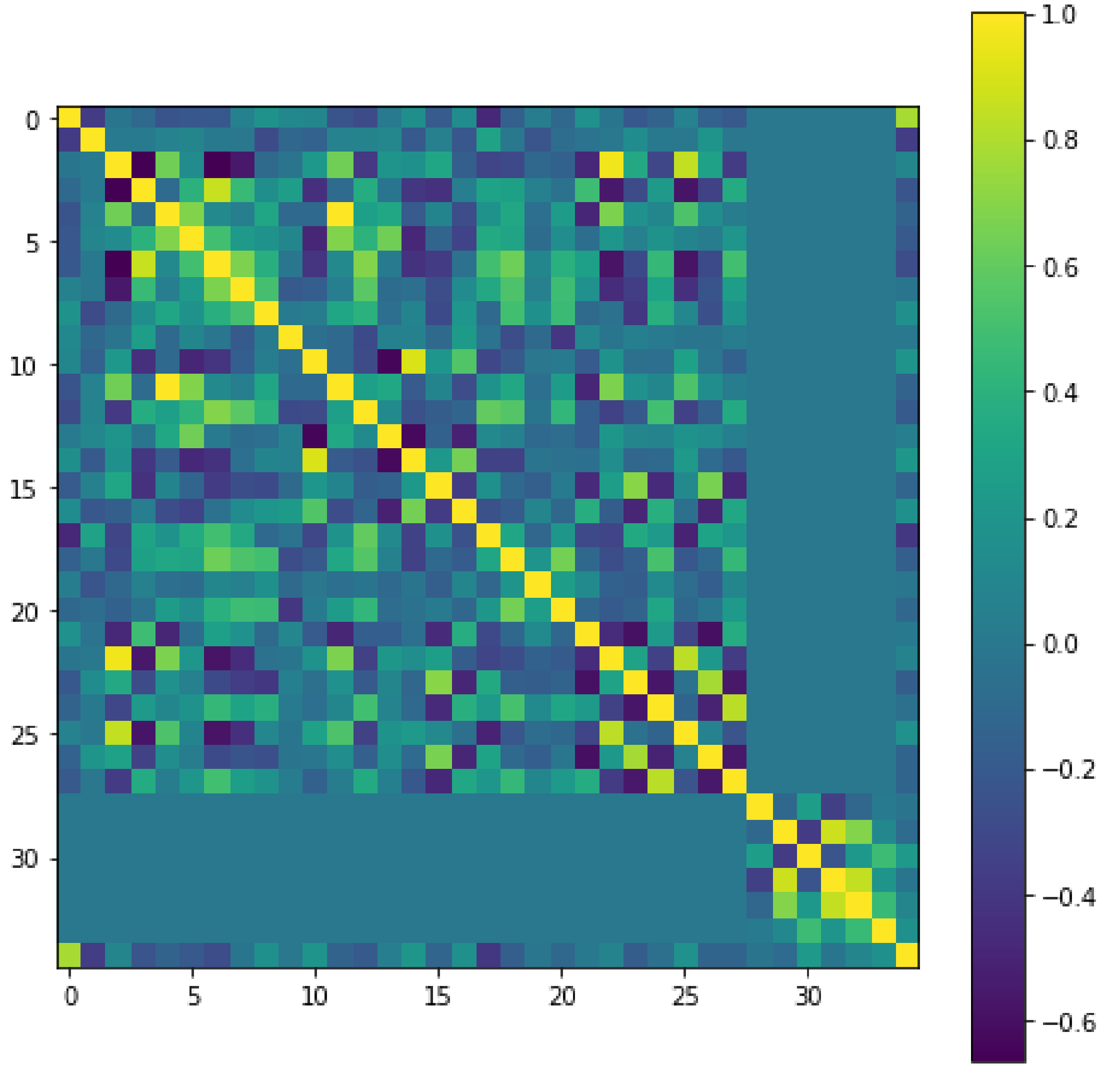


The colorbar indicates that the orders of magnitude of different features are quite different, thus it may be necessary to scale the features to help improve regression performance.

```
In [10]: X_scaled = (X - X.mean(axis=0))/X.std(axis=0)
         covar = np.cov(X_scaled.T)
```

```
        fig,ax = plt.subplots(figsize=(8, 8))
        c = ax.imshow(covar)
        fig.colorbar(c)
```

```
Out[10]: <matplotlib.colorbar.Colorbar at 0x1182f9198>
```



The scaled heat map shows an interesting property: Feature 0-28 and Feature 29-34 are independent. This makes sense as the former is the features for MOFs and the latter is the ones for adsorbates.

1.8 Train-test-validation set split

```
In [11]: np.random.seed(5)
         from sklearn.model_selection import train_test_split
```

```
# combine the unscaled and scaled X, so that they can be split together
X_combined = np.concatenate((X, X_scaled), axis=1)

# ------------------------------------------------------------------------------
# ---------------------------- don't touch the validation set ------------------------
X_train_test_combined, X_validation_combined, y_train_test, y_validation = train_test

X_validation, X_validation_scaled = X_validation_combined[:, :35], X_validation_combi
# ---------------------------- don't touch the validation set ------------------------
# ------------------------------------------------------------------------------

X_train_combined, X_test_combined, y_train, y_test = train_test_split(X_train_test_co
                                                                       y_train_test, t
X_train, X_train_scaled = X_train_combined[:, :35], X_train_combined[:, 35:]
X_test, X_test_scaled = X_test_combined[:, :35], X_test_combined[:, 35:]
```

2 Baseline models

Two models are first trained as baseline models: multi-linear regression and kernel regression.

2.1 Multi-linear regression

2.1.1 Regression using unscaled data

```
In [12]: from sklearn.linear_model import LinearRegression

         LR_model = LinearRegression()
         LR_model.fit(X_train, y_train)
         r2_LR_train = LR_model.score(X_train, y_train)
         r2_LR_test = LR_model.score(X_test, y_test)

         y_hat_train = LR_model.predict(X_train)
         y_hat_test = LR_model.predict(X_test)

         print("r^2 train:\t{:.4f}".format(r2_LR_train))
         print("r^2 test:\t{:.4f}".format(r2_LR_test))
```

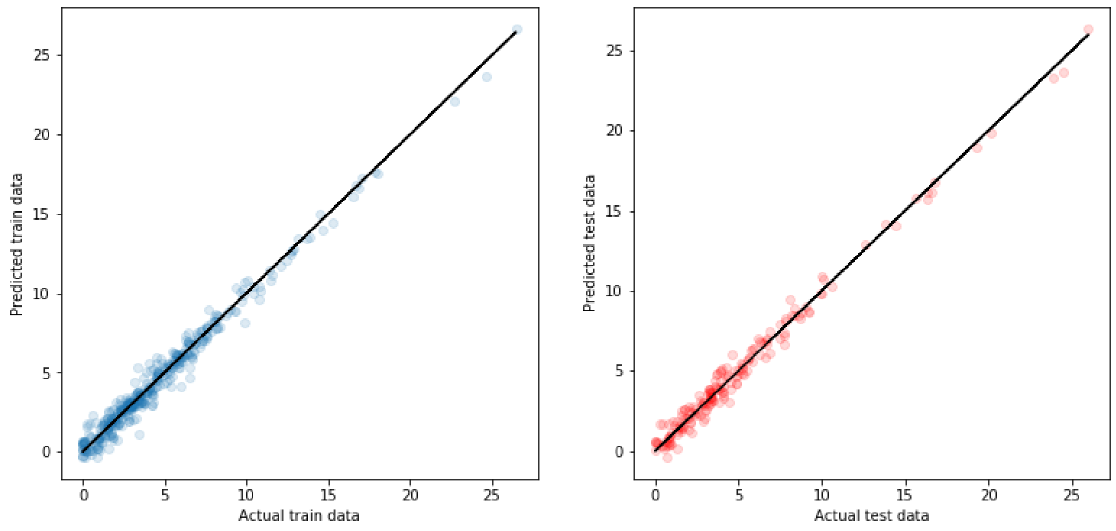
```
r^2 train:        0.9790
r^2 test:        0.9864
```

With high $r^2 (> 0.97)$ for both trainning set and test set, the multi-linear model predicts data well.

```
In [13]: fig, axes = plt.subplots(1, 2, figsize=(13, 6))

         axes[0].scatter(y_train, y_hat_train, alpha=0.15)
```

```
        axes[0].plot(y_train, y_train, '-k')
        axes[0].set_xlabel('Actual train data')
        axes[0].set_ylabel('Predicted train data')

        axes[1].scatter(y_test, y_hat_test, alpha=0.15, c='r')
        axes[1].plot(y_test, y_test, '-k')
        axes[1].set_xlabel('Actual test data')
        axes[1].set_ylabel('Predicted test data')
```

```
Out[13]: Text(0, 0.5, 'Predicted test data')
```



From the parity plots for train set, we found that the outliers are uniformly distributed around the line. The model works with both sets well.

2.1.2 Regression using scaled data

```
In [14]: LR_model_scaled = LinearRegression()
         LR_model_scaled.fit(X_train_scaled, y_train)

         r2_LR_train_scaled = LR_model_scaled.score(X_train_scaled, y_train)
         r2_LR_test_scaled = LR_model_scaled.score(X_test_scaled, y_test)

         y_hat_train_scaled = LR_model_scaled.predict(X_train_scaled)
         y_hat_test_scaled = LR_model_scaled.predict(X_test_scaled)

         print("r^2 train:\t{:.4f}".format(r2_LR_train_scaled))
         print("r^2 test:\t{:.4f}".format(r2_LR_test_scaled))
```

```
r^2 train:       0.9790
r^2 test:        0.9864
```

By normalizing the data, we didn't see improvement on the r^2.

2.2 Multi-linear regression

2.2.1 Regression using unscaled data

```
In [15]: from sklearn.metrics.pairwise import rbf_kernel

         # set up different paprameters
         sigmas = np.array([1E-4, 5E-4, 1E-3, 5E-3, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 15, 20, 25
         gammas = 1./(2*sigmas**2)
         r2_matrix = np.zeros((gammas.size, 3))

         # calculate r2 enumerating the list of gammas
         for i, gamma in enumerate(gammas):

             # create rbf with given gamma
             X_train_kernel = rbf_kernel(X_train, X_train, gamma = gamma)
             X_test_kernel = rbf_kernel(X_test, X_train, gamma = gamma)
             model_rbf = LinearRegression()
             model_rbf.fit(X_train_kernel, y_train)

             # record the gamma for the best prediction and the corresponding r2
             r2_matrix[i, 0] = gamma
             r2_matrix[i, 1] = model_rbf.score(X_train_kernel, y_train)
             r2_matrix[i, -1] = model_rbf.score(X_test_kernel, y_test)

         # find the best r2
         n = r2_matrix[:, -1].argmax()
         print("Best gamma:\t{:g} (sigma = {:n})".format(r2_matrix[n, 0], (1 / 2 / r2_matrix[n
         print("r^2 train:\t{:.4f}".format(r2_matrix[n, 1]))
         print("r^2 test:\t{:.4f}".format(r2_matrix[n, -1]))
```

```
Best gamma:         50 (sigma = 0.1)
r^2 train:         1.0000
r^2 test:         0.0146
```

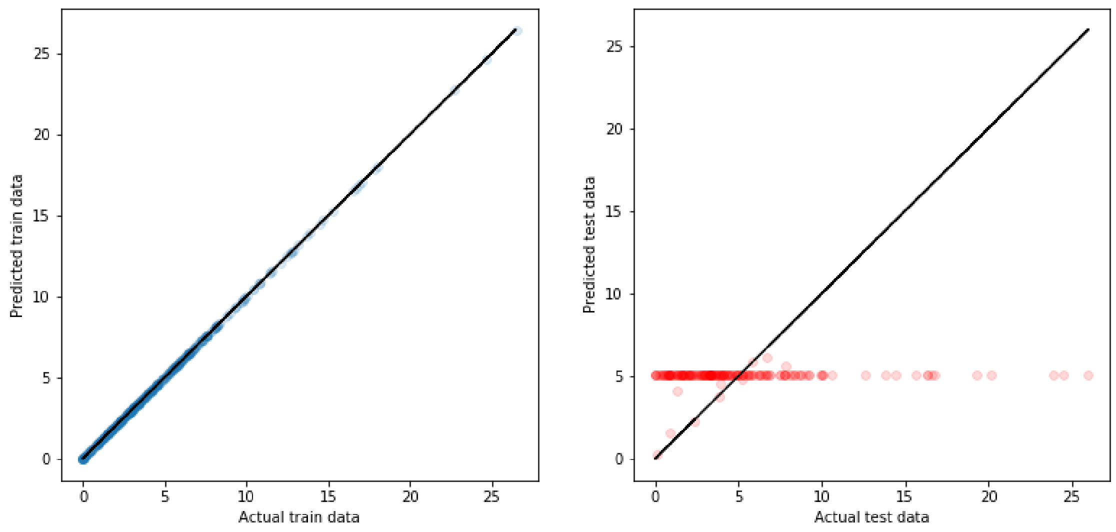
The big difference between the r2 scores of the training set and the test set is very interesting. To investigate if it is the common case for all gamma values, the r2 matrix is printed below (columns are: gamma, r2 for training, r2 to test):

```
In [16]: np.set_printoptions(precision=3,suppress=True)
         print(r2_matrix)
```

```
[[50000000.           1.          -0.01 ]
 [ 2000000.           1.          -0.01 ]
 [  500000.           1.          -0.015]
 [   20000.           1.          -0.01 ]
 [    5000.           1.          -0.004]
 [     200.           1.           0.005]
 [      50.           1.           0.015]
 [       2.           1.          -0.185]
 [       0.5          1.     -315380.541]
 [       0.02         1.        -198.026]
 [       0.005        1.        -172.691]
 [       0.002        1.        -133.9  ]
 [       0.001        1.        -241.966]
 [       0.001        1.        -832.324]
 [       0.001        1.       -1758.694]
 [       0.           1.       -1551.724]
 [       0.           1.        -350.012]
 [       0.           1.          -4.596]]
```

The table confirmed our assumption. This indicates that the kernel model overfits with all the given gamma values, which is also shown in the parity plots below:

```
In [17]: best_gamma = r2_matrix[n, 0]
         X_train_kernel = rbf_kernel(X_train, X_train, gamma = best_gamma)
         model_rbf = LinearRegression()
         model_rbf.fit(X_train_kernel, y_train)
         y_hat_train = model_rbf.predict(X_train_kernel)
         X_test_kernel = rbf_kernel(X_test, X_train, gamma = best_gamma)
         y_hat_test = model_rbf.predict(X_test_kernel)

         fig, axes = plt.subplots(1, 2, figsize=(13, 6))

         axes[0].scatter(y_train, y_hat_train, alpha=0.15)
         axes[0].plot(y_train, y_train, '-k')
         axes[0].set_xlabel('Actual train data')
         axes[0].set_ylabel('Predicted train data')

         axes[1].scatter(y_test, y_hat_test, alpha=0.15, c='r')
         axes[1].plot(y_test, y_test, '-k')
         axes[1].set_xlabel('Actual test data')
         axes[1].set_ylabel('Predicted test data')

Out[17]: Text(0, 0.5, 'Predicted test data')
```



The right plot clearly shows that the model failed to predict a lot of data points (notice the horizontal line around 5-the model predicts this value for many data points, indicating that the model does not capture some relationships among features.)

The gamma value can be further refined around the best value (50):

```
In [18]: gammas = np.linspace(10, 100)

         r2_matrix = np.zeros((gammas.size, 3))

         # calculate r2 enumerating the list of gammas
         for i, gamma in enumerate(gammas):

             # create rbf with given gamma
             X_train_kernel = rbf_kernel(X_train, X_train, gamma = gamma)
             X_test_kernel = rbf_kernel(X_test, X_train, gamma = gamma)
             model_rbf = LinearRegression()
             model_rbf.fit(X_train_kernel, y_train)

             # record the gamma for the best prediction and the corresponding r2
             r2_matrix[i, 0] = gamma
             r2_matrix[i, 1] = model_rbf.score(X_train_kernel, y_train)
             r2_matrix[i, -1] = model_rbf.score(X_test_kernel, y_test)

         # find the best r2
         n = r2_matrix[:, -1].argmax()
         print("Best gamma:\t{:g} (sigma = {:n})".format(r2_matrix[n, 0], (1 / 2 / r2_matrix[n
         print("r^2 train:\t{:.4f}".format(r2_matrix[n, 1]))
         print("r^2 test:\t{:.4f}".format(r2_matrix[n, -1]))
```

```
Best gamma:        33.8776 (sigma = 0.121487)
r^2 train:        1.0000
```

```
r^2 test:        0.0167
```

The refined gamma does not improve the result much.

2.2.2 Regression using scaled data

```
In [19]: sigmas = np.array([1E-4, 5E-4, 1E-3, 5E-3, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 15, 20, 25
         gammas = 1./(2*sigmas**2)
         r2_matrix = np.zeros((gammas.size, 3))

         for i, gamma in enumerate(gammas):

             # create rbf with given gamma, using scaled data
             X_train_kernel_scaled = rbf_kernel(X_train_scaled, X_train_scaled, gamma = gamma)
             X_test_kernel_scaled = rbf_kernel(X_test_scaled, X_train_scaled, gamma = gamma)
             model_rbf_scaled = LinearRegression()
             model_rbf_scaled.fit(X_train_kernel_scaled, y_train)

             r2_matrix[i, 0] = gamma
             r2_matrix[i, 1] = model_rbf.score(X_train_kernel_scaled, y_train)
             r2_matrix[i, -1] = model_rbf.score(X_test_kernel_scaled, y_test)

         n = r2_matrix[:, -1].argmax()
         print("Best gamma:\t{:g} (sigma = {:n})".format(r2_matrix[n, 0], (1 / 2 / r2_matrix[n
         print("r^2 train:\t{:.4f}".format(r2_matrix[n, 1]))
         print("r^2 test:\t{:.4f}".format(r2_matrix[n, -1]))
         best_gamma = r2_matrix[n, 0]
```

```
Best gamma:        0.5 (sigma = 1)
r^2 train:        0.5599
r^2 test:        0.3449
```

```
In [20]: np.set_printoptions(precision=3,suppress=True)
         print(r2_matrix)
```

```
[[50000000.          -0.041       -0.009]
 [ 2000000.          -0.041       -0.009]
 [  500000.          -0.041       -0.009]
 [   20000.          -0.041       -0.009]
 [    5000.           0.009       -0.008]
 [     200.           0.978        0.005]
 [      50.           0.99         0.007]
 [       2.           0.962        0.167]
 [       0.5          0.56         0.345]
 [       0.02      -282.874     -205.621]
 [       0.005     -219.257     -158.034]
 [       0.002     -147.571     -102.115]
 [       0.001     -119.767      -81.202]
 [       0.001     -107.396      -72.179]
 [       0.001     -100.994      -67.618]
 [       0.         -94.947      -63.405]
 [       0.         -92.279      -61.584]
 [       0.         -90.87       -60.633]]
```

For the scaled data, the kernel model does not overfit for the training set, but the score is still not satisfactory.

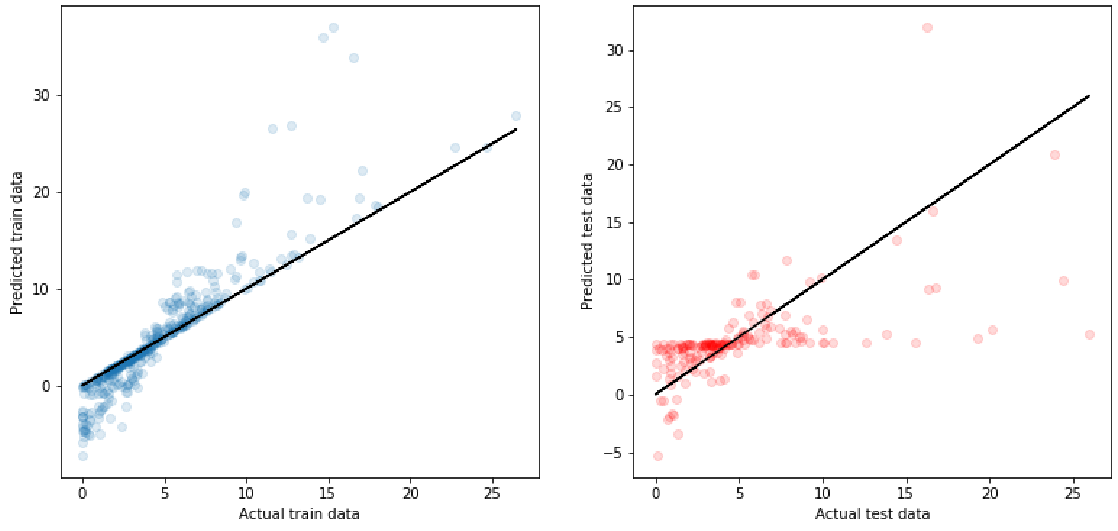
The gamma value can be further refined around the best value (0.5):

```
In [21]: gammas = np.linspace(0.02, 2, 100)
         r2_matrix = np.zeros((gammas.size, 3))

         for i, gamma in enumerate(gammas):

             # create rbf with given gamma, using scaled data
             X_train_kernel_scaled = rbf_kernel(X_train_scaled, X_train_scaled, gamma = gamma)
             X_test_kernel_scaled = rbf_kernel(X_test_scaled, X_train_scaled, gamma = gamma)
             model_rbf_scaled = LinearRegression()
             model_rbf_scaled.fit(X_train_kernel_scaled, y_train)

             r2_matrix[i, 0] = gamma
             r2_matrix[i, 1] = model_rbf.score(X_train_kernel_scaled, y_train)
             r2_matrix[i, -1] = model_rbf.score(X_test_kernel_scaled, y_test)

         n = r2_matrix[:, -1].argmax()
         print("Best gamma:\t{:g} (sigma = {:n})".format(r2_matrix[n, 0], (1 / 2 / r2_matrix[n
         print("r^2 train:\t{:.4f}".format(r2_matrix[n, 1]))
         print("r^2 test:\t{:.4f}".format(r2_matrix[n, -1]))
         best_gamma = r2_matrix[n, 0]

Best gamma:        0.44 (sigma = 1.066)
r^2 train:        0.4305
r^2 test:        0.3490
```

```
In [22]: best_gamma = r2_matrix[n, 0]
         X_train_kernel_scaled = rbf_kernel(X_train_scaled, X_train_scaled, gamma = best_gamma
         X_test_kernel_scaled = rbf_kernel(X_test_scaled, X_train_scaled, gamma = best_gamma)
         model_rbf_scaled = LinearRegression()
         model_rbf_scaled.fit(X_train_kernel_scaled, y_train)
         y_hat_train = model_rbf.predict(X_train_kernel_scaled)
         y_hat_test = model_rbf.predict(X_test_kernel_scaled)

         fig, axes = plt.subplots(1, 2, figsize=(13, 6))

         axes[0].scatter(y_train, y_hat_train, alpha=0.15)
```

```
axes[0].plot(y_train, y_train, '-k')
axes[0].set_xlabel('Actual train data')
axes[0].set_ylabel('Predicted train data')

axes[1].scatter(y_test, y_hat_test, alpha=0.15, c='r')
axes[1].plot(y_test, y_test, '-k')
axes[1].set_xlabel('Actual test data')
axes[1].set_ylabel('Predicted test data')
```

```
Out[22]: Text(0, 0.5, 'Predicted test data')
```



2.3 Conclusion for baseline models

We have tested two models for this dataset: multi-linear model and kernel regression model (using rbf). - The multi-linear model manages to give high-quality regression results with both unscaled and scaled data. - We tried to apply simple hyperparameter tuning strategies for the kernel regression model, but the kernel model tends to overfit with the unscaled training set with all gamma values. When it is trained with unscaled training set, the overfitting issue no longer exists, but the model performs poorly for both the training and the test set.

The difference of the performance of two models may indicate that this data set works well with linear models, but may suffer severely with kernel based nonlinear model. This is a very interesting property that we have not seen in the examples given in the class. Our preliminary explanation is that this is caused by the combination of MOF features and adsorbate features (see the analysis in Section 1.7). Since two feature groups are independent of each other, the kernel that we created actually accounts for interaction between these two groups that does not exists, which causes the poor performance of the kernel model. We would like to further discuss with the instructors to figure out the reason behind this phenomena.

---

3 Next steps

1. investigate the cause of the poor performance of the kernel model
2. apply other models (kernel ridge regression, LASSO, neuron network, etc.) for regression
3. apply principal component analysis to see if regression based on principal components will have better performance
4. assess the performance of all models using the validation set