ChBE 6745 Project: Improved Model report

Prediction of Adsorption Properties of Metal-Organic Frameworks with Framework Flexibility

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

April 3, 2020

Based on our project plan and the feedback on our baseline models, we focus mainly on the following points in the improved model report:

- We address the **issue of the initial validation strategy** and **present the new validation strategy**, in Section 2.3 and Section 4.
- We **analyze the overfitting issue of the pure rbf model**, and **re-implement it using the new validation strategy** in Section 5.
- We **implement three new models** on the dataset: **KRR, LASSO, MLPRegressor (neural network)** in Section 6-8.
- We add more descriptions on our project goals (Section 1.1), dataset, model training strategy and model validation strategy (Section 2.1-2.3).

1 Introduction

1.1 Project description

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). Involving framework flexibility in molecular simulation may significantly affect adsorption uptakes and selectivities. However, taking the framework flexibility into account directly in the molecular simulation framework may be 5-10 times more computationally expensive than standard simulation methods based on rigid crystal structure. Therefore, we are interested in constructing a regression model to predict the adsorption properties of MOFs with framework flexibility involved.

1.2 Goals

We plan to construct different regression models and evaluate their performance on **predicting single-component adsorption uptake of MOFs with framework flexibility with respect to certain adosrbates at saturation pressure**, based on three types of input:

1. the features of the MOFs,
2. the features of adsorbates, and
3. the single-component adsorption uptake of MOFs with framework flexibility with respect to certain adosrbates *from standard simulation methods based on rigid crystal structure.*

The main focus of this project is to:

1. implement various regression algorithms on the proposed dataset to find the one with the best prediction ability,
2. apply hyperparameter tuning for each algorithm to improve model performance, and
3. combine different validation strategies and apply throughout the project to make the models robust and least biased.

The features of the MOF have been pre-selected and the number of the overall features is relatively small, so we do not address feature engineering as the main focus of our project except for the LASSO models.

2 Methodology

2.1 Dataset details

The dataset discussed below is provided by Dr. David Sholl's group. The corresponding files can be found at the `data` folder.

The dataset is composed of three parts:

1. **28 MOF features** (under `data/ML_data`),
2. **6 adsorbate features** (manually added below), and
3. **adsorption uptakes** of **801 (MOF, adsorbate) pairs** (under `data/flexibility_data/y_data/adsorption_data`), containing two values:
   (a) values from rigid model (serve as a feature)
   (b) mean values from flexible model (serve as y)

All features are continuous entries. The dimension of the input is $(801, 28 + 6 + 1) = (801, 35)$. The dimension of the output is $(801, 1)$.

2.2 Model training strategy

We plan to train 5 models for the proposed problem:

1. Multi-linear regression
2. RBF kernel regression
3. Kernel ridge regression (KRR)
4. LASSO regression
5. MLPRegressor (neural network)

Multi-linear model serves as a baseline model to assess the performance of other models. RBF kernel regression (without regularization) also serves as a baseline model with nonlinearity. These two models were built and tested in the baseline model report.

The rest three models are covered in this report. The KRR model is built to show the power of regularization compared with the pure RBF kernel regression model. The LASSO model is built for both multi-linear regression and RBF kernel regression. It can be used to see the impact of complexity optimization on the prediction performance. Finally the MLPRegressor model is built as a representation of the neural network algorithms. Implementing this model is an attempt on models that are not covered in the class and will be tested against other algorithms in the final report.

2.3 Model validation strategy

Our *new* validation strategy is as follows: 1. We first apply hold-out to split 25% of the data as validation set, which would not be modified or changed throughout the training-testing process. 2. After hold-out, we then apply 5-fold on the rest 75% data to form five (training set, test set) pairs. Each training set contains 75%*80%=60% data, and each test set contains 75%*20%=15% data. The same five pairs are used throughout the project for model training and hyperparameter tuning for all models. In this way the training and the tuning processes will be least biased by how the dataset is split into the training and test sets.

The 5-fold is implemented based on the following feature of `GridSearchCV`: it can take CV splitter object as its `cv` argument. Therefore, if we can define a 5-fold CV splitter object beforehand and apply it during the training process for all the models, we can guarantee that the same (training set, test set) pairs are used and no bias is generated by the validation strategy. For models that do not use `GridSearchCV`, an explicit for-loop iterating the five (training set, test set) pairs is used instead, which are generated from the same CV splitter object.

In the baseline model report, a relatively simple strategy is used: after hold-out to split 25% of the data as validation set, we applied another hold-out to split the rest data into a single (training set, test set) pair. The latter can be seen as applying a 1-fold. Compared with the current strategy, the previous validation strategy is significantly less robust, as it is highly possible to be affected by the sampling of training/test set, while the 5-fold can alleviate this issue. Therefore, we can expect that the models trained with the new validation strategy will be more stable in terms of prediction quality.

The implementation the new validation strategy is shown in Section 4.

2.4 Hyperparameter optimization strategy

A general hyperparameter optimization strategy is used for all models except for `MLPRegressor`, which is described below:

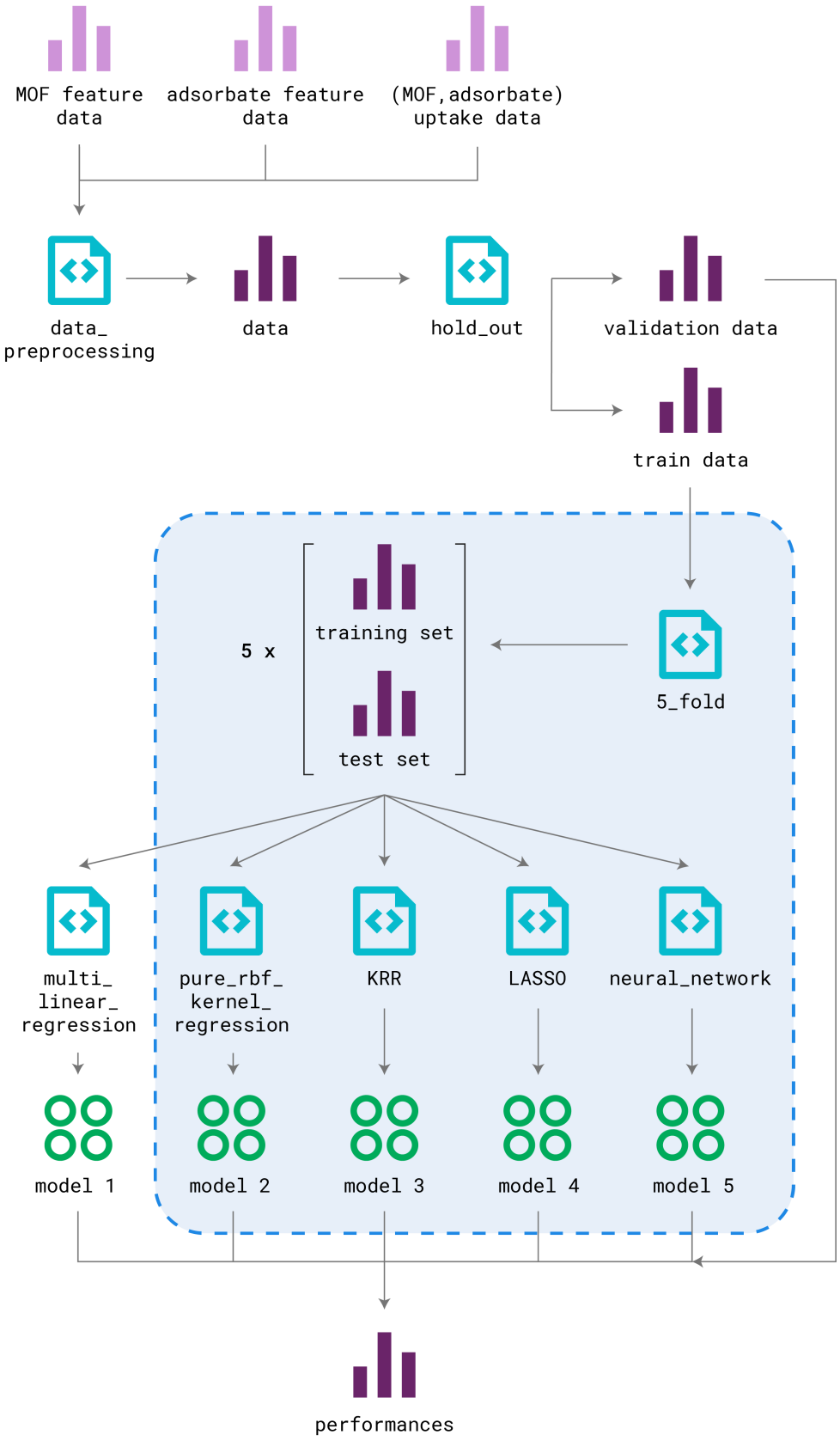
1. rough estimation: for each hyperparameter, a list of value range (typically roughly logarithmically discretized) is given to `GridSearchCV`. `GridSearchCV` searchs through the lists for all hyperparameters and returns the best ones.

2. refined estimation: for each hyperparameter, a list of value range close to the best value determined in the first step is given to `GridSearchCV`. `GridSearchCV` searchs through the lists for all hyperparameters and returns the best ones.

This strategy assumes that the loss function is convex over all hyperparameters jointly. This may be not reasonable but it will be well beyond the scope of this course to address this issue, so we apply this strategy for all models except `MLPRegressor`, whose tuning strategy is discussed separately in Section 7.1.

2.5 Block diagram of data pipeline and model(s)

The block diagram is given below, where the highlighted part is covered in the improved model report.


MOF feature data
adsorbate feature data
(MOF,adsorbate) uptake data
data_ preprocessing
data
hold_out
validation data
train data
5 x
training set
test set
5_fold
multi_ linear_ regression
pure_rbf_ kernel_ regression
KRR
LASSO
neural_network
model 1
model 2
model 3
model 4
model 5
performances

2.6 Software packages

All algorithms are imported from `sklearn`.

- multi-linear regression: `sklearn.linear_model.LinearRegression`
- rbf kernel regression: `sklearn.metrics.pairwise.rbf_kernel`
- KRR: `sklearn.kernel_ridge.KernelRidge`
- LASSO: `sklearn.linear_model.Lasso`
- MLPregressor: `sklearn.neural_network.MLPRegressor`
- validation strategy: `sklearn.model_selection.train_test_split`, `sklearn.model_selection.KFold`
- hyperparameter tuning: `sklearn.model_selection.GridSearchCV`

3 Data preprocessing

This section is the same as the one in the baseline model report, so no extra explanation or comment is provided here.

In [1]:
```
# Preamble for package import and figure formatting

import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
from sklearn.kernel_ridge import KernelRidge
from sklearn.neural_network import MLPRegressor
from sklearn.metrics.pairwise import rbf_kernel

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'png')
plt.rcParams['savefig.dpi'] = 75
plt.rcParams['figure.autolayout'] = False
plt.rcParams['figure.figsize'] = 10, 6
plt.rcParams['axes.labelsize'] = 18
plt.rcParams['axes.titlesize'] = 20
plt.rcParams['font.size'] = 16
plt.rcParams['lines.linewidth'] = 2.0
plt.rcParams['lines.markersize'] = 8
plt.rcParams['legend.fontsize'] = 14
plt.rcParams['font.family'] = "serif"
plt.rcParams['font.serif'] = "cm"
# plt.rcParams['text.usetex'] = True
# plt.rcParams['text.latex.preamble']= r"\usepackage{subdepth}, \usepackage{type1cm}"
```

In [2]:
```
# read MOF feature data

df36Descriptor = pd.read_excel('data/ML_data/descriptor_used.xlsx',header=4,index_col=1)

# 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]
```

In [3]:
```
# Read adsorption update data

# the MOFs in "dfMLReduced" and adsorption data sets are different, so it is necessary t
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 list
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)

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

In [4]:
```
# manually add adsorbate descriptors

# Mw/gr.mol-1, Tc/K, Pc/bar, ω, 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]])

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

# 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"]
```

In [5]:
```
## Combine MOF and adsorbate feature data

# 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)
```

In [6]:
```
        # generate X and y

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

        # feature scaling
        X_scaled = (X - X.mean(axis=0))/X.std(axis=0)
```

4 Validation strategy

4.1 Hold-out for validation set

In [54]:
```
np.random.seed(5)

        # 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_s

        X_train_test, X_train_test_scaled = X_train_test_combined[:, :35], X_train_test_combine
        X_validation, X_validation_scaled = X_validation_combined[:, :35], X_validation_combine
        # ---------------------------- don't touch the validation set ------------------------
```

4.2 5-Fold for model training

As discussed in Section 2.3, we utilize the feature of `GridSearchCV` applying the same CV-splitter object `kf` for all models for training and hyperparameter optimization, which is defined below.

In [55]:
```
        # create a CV-splitter object that is used for all models
        kf = KFold(n_splits=5, shuffle = True, random_state=20)
        # kf = KFold(n_splits=5, shuffle = True)

        # for models that cannot use kf, an explicit (train, test) pairs are created
        for i, (train_index, test_index) in enumerate(kf.split(X_train_test_combined)):

            # initialize sets
            if i == 0:
                X_train_combined_5fold = np.zeros(X_train_test_combined[train_index].shape + (5
                X_test_combined_5fold = np.zeros(X_train_test_combined[test_index].shape + (5,)
```

```
            y_train_5fold = np.zeros(y_train_test[train_index].shape + (5,))
            y_test_5fold = np.zeros(y_train_test[test_index].shape + (5,))

        X_train_combined_5fold[:, :, i], X_test_combined_5fold[:, :, i] = X_train_test_comb
        y_train_5fold[:, :, i], y_test_5fold[:, :, i] = y_train_test[train_index], y_train_

    X_train_5fold, X_train_scaled_5fold = X_train_combined_5fold[:, :35], X_train_combined_
    X_test_5fold, X_test_scaled_5fold = X_test_combined_5fold[:, :35], X_test_combined_5fol
```

5 Re-implementing RBF kernel regression (without regularization)

As mentioned in Section 2.3, we used a single 1-fold validation strategy when training the RBF kernel regression model in the baseline model report. The model obtained $r^2 = 0.0167$ with unscaled data and $r^2 = 0.3449$ with scaled data, both of which are significantly worse than the ones obtained from the multi-linear model. The main reason for the poor performance is that 1. The original 1-fold strategy led to extremely umbalanced sampling between the training and the testing set. This can be avoided by updating the validation strategy. 2. The overfitting to training sets is unavoidable. So regularization is necessary for the model.

We address the first point by re-implementing the RBF kernel regression using our new validation strategy. For the second point, we would like to compare the results from RBF kernel regression with results from KRR to show the power of regularization in the final report.

Since pure RBF kernel regression model cannot directly apply `GridSearchCV`, we manually apply hyperparameter tuning using for-loops. We basically follow the general hyperparameter tuning strategy described in Section 2.4. After the best γ is obtained in the rough estimation for a certain fold, we directly apply refined estimation in that certain fold. The pure RBF kernel model is trained on both the unscaled data and the scaled data.

5.1 Regression using unscaled data

In [56]:
```
# 1st round hyperparameter tuning

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))

rbf_r2_list = []
rbf_coef_list = []
rbf_gamma_list = []

for i in range(5):

    rbf_coef_temp_list = [] # temporarily store parameter values for each gamma

    for j, gamma in enumerate(gammas):

        X_train_kernel = rbf_kernel(X_train_5fold[:, :, i], X_train_5fold[:, :, i], gam
        X_test_kernel = rbf_kernel(X_test_5fold[:, :, i], X_train_5fold[:, :, i], gamma
        model_rbf = LinearRegression()
```

```
            model_rbf.fit(X_train_kernel, y_train_5fold[:, :, i])

            r2_matrix[j, 0] = gamma
            r2_matrix[j, 1] = model_rbf.score(X_train_kernel, y_train_5fold[:, :, i])
            r2_matrix[j, -1] = model_rbf.score(X_test_kernel, y_test_5fold[:, :, i])

            rbf_coef_temp_list.append(model_rbf.coef_)

        # For each fold, record the optimal gamma and the corresponding parameter values an
        n = r2_matrix[:, -1].argmax()
        rbf_r2_list.append(r2_matrix[n, -1])
        rbf_gamma_list.append(r2_matrix[n, 0])
        rbf_coef_list.append(rbf_coef_temp_list[n])


    best_i = np.array(rbf_r2_list).argmax()
    rbf_r2_best1 = rbf_r2_list[best_i]
    rbf_gamma_best = rbf_gamma_list[best_i]
    rbf_coef_best = rbf_coef_list[best_i]


    for i in range(5):
        if i != best_i:
            print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}".format(i, rbf_r2_list[i], rb
        else:
            print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}\t\t<-best performance".format
```

```
Fold 0:     r2=0.048,       optimal gamma=2e+00
Fold 1:     r2=0.314,       optimal gamma=2e-03             <-best performance
Fold 2:     r2=0.019,       optimal gamma=2e+02
Fold 3:     r2=0.028,       optimal gamma=2e+02
Fold 4:     r2=0.021,       optimal gamma=2e+02
```

In [57]:
```
refined_gammmas = np.append(np.logspace(np.log(rbf_gamma_best) - 1, np.log(rbf_gamma_be

r2_matrix = np.zeros((gammas.size, 3))
rbf_coef_temp_list = []

for j, gamma in enumerate(refined_gammmas):

    X_train_kernel = rbf_kernel(X_train_5fold[:, :, best_i], X_train_5fold[:, :, best_i
    X_test_kernel = rbf_kernel(X_test_5fold[:, :, best_i], X_train_5fold[:, :, best_i],
    model_rbf = LinearRegression()
    model_rbf.fit(X_train_kernel, y_train_5fold[:, :, best_i])

    r2_matrix[j, 0] = gamma
    r2_matrix[j, 1] = model_rbf.score(X_train_kernel, y_train_5fold[:, :, best_i])
```

```
            r2_matrix[j, -1] = model_rbf.score(X_test_kernel, y_test_5fold[:, :, best_i])

            rbf_coef_temp_list.append(model_rbf.coef_)

        n = r2_matrix[:, -1].argmax()
        rbf_refined_r2 = r2_matrix[n, -1]
        rbf_refined_gamma = r2_matrix[n, 0]
        rbf_refined_coef = rbf_coef_temp_list[n]

        print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}".format(best_i, rbf_refined_r2, rbf_r
```

```
Fold 1:         r2=0.829,         optimal gamma=8e-08
```

5.2 Regression using scaled data

```
In [58]: 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))

         rbf_r2_list_scaled = []
         rbf_coef_list_scaled = []
         rbf_gamma_list_scaled = []

         for i in range(5):

             rbf_coef_temp_list = []

             for j, gamma in enumerate(gammas):

                 X_train_scaled_kernel = rbf_kernel(X_train_scaled_5fold[:, :, i], X_train_scale
                 X_test_scaled_kernel = rbf_kernel(X_test_scaled_5fold[:, :, i], X_train_scaled_
                 model_rbf = LinearRegression()
                 model_rbf.fit(X_train_scaled_kernel, y_train_5fold[:, :, i])

                 r2_matrix[j, 0] = gamma
                 r2_matrix[j, 1] = model_rbf.score(X_train_scaled_kernel, y_train_5fold[:, :, i]
                 r2_matrix[j, -1] = model_rbf.score(X_test_scaled_kernel, y_test_5fold[:, :, i])

                 rbf_coef_temp_list.append(model_rbf.coef_)

             n = r2_matrix[:, -1].argmax()
             rbf_r2_list_scaled.append(r2_matrix[n, -1])
             rbf_gamma_list_scaled.append(r2_matrix[n, 0])
             rbf_coef_list_scaled.append(rbf_coef_temp_list[n])


         best_i = np.array(rbf_r2_list_scaled).argmax()
```

```
        rbf_r2_best_scaled = rbf_r2_list_scaled[best_i]
        rbf_gamma_best_scaled = rbf_gamma_list_scaled[best_i]
        rbf_coef_best_scaled = rbf_coef_list_scaled[best_i]

        for i in range(5):
            if i != best_i:
                print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}".format(i, rbf_r2_list_scaled
            else:
                print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}\t\t<-best performance".format
```

```
Fold 0:     r2=0.951,       optimal gamma=2e-02             <-best performance
Fold 1:     r2=0.082,       optimal gamma=8e-04
Fold 2:     r2=0.175,       optimal gamma=2e+00
Fold 3:     r2=0.123,       optimal gamma=2e+00
Fold 4:     r2=0.504,       optimal gamma=5e-03
```

```
In [59]: scaled_refined_gammmas = np.append(np.logspace(np.log(rbf_gamma_best_scaled) - 1, np.lo
         r2_matrix = np.zeros((scaled_refined_gammmas.size, 3))
         rbf_coef_temp_list = []

         for j, gamma in enumerate(scaled_refined_gammmas):

             X_train_scaled_kernel = rbf_kernel(X_train_scaled_5fold[:, :, best_i], X_train_scal
             X_test_scaled_kernel = rbf_kernel(X_test_scaled_5fold[:, :, best_i], X_train_scaled
             model_rbf = LinearRegression()
             model_rbf.fit(X_train_scaled_kernel, y_train_5fold[:, :, best_i])

             r2_matrix[j, 0] = gamma
             r2_matrix[j, 1] = model_rbf.score(X_train_scaled_kernel, y_train_5fold[:, :, best_i
             r2_matrix[j, -1] = model_rbf.score(X_test_scaled_kernel, y_test_5fold[:, :, best_i]

             rbf_coef_temp_list.append(model_rbf.coef_)

         n = r2_matrix[:, -1].argmax()
         rbf_refined_r2 = r2_matrix[n, -1]
         rbf_refined_gamma = r2_matrix[n, 0]
         # rbf_refined_coef = rbf_coef_temp_list[n]

         print("Fold {}:\tr2={:.3f},\toptimal gamma={:.0e}".format(best_i, rbf_refined_r2, rbf_r
```

```
Fold 0:     r2=0.951,       optimal gamma=2e-02
```

5.3 Discussion

By updating the validation strategy, we managed to significantly improve the performance of the pure RBF kernel regression model):

r^2	original validation strategy	new validation strategy
unscaled data	0.017	0.829
scaled data	0.345	0.951

However, it is worth noting that RBF kernel regression is not a stable method for the dataset. The hyperparameter tuning processes above show that the model performs poorly for some folds of the train-test sets, and is only good for certain folds, indicating that the model is very sensitive to the split of the train-test sets. When the `random_state` argument of `kf` is changed, it is observed that the performance of the model is changing drastically and the r^2 values may not be as good as the ones in the table (although they are still better than the one obtained using the original validation strategy). This shows the necessity of involving regularization into the model, which motivates the implementation of KRR models in the next section.

6 Kernel ridge regression (KRR)

Regularization is especially important in non-parametric models. In order to avoid over-fitting the model, penalizing models that change very sharply is necessary. By adding a penalty for very large parameters in the loss function, the error of the model is kept small while the size of the parameters is also kept small. By using a kernel and regularize on the sum of squared parameters it is called Kernel Ridge Regression, or KRR. In this project, we use **radial basis function** as the kernel in KRR. The loss function for KRR is as follows:

$$\mathcal{L}_{\text{KRR}} = \sum_i \varepsilon_i^2 + \alpha||\vec{w}||_2$$

6.1 Hyperparameter tuning strategy

There are two main hyperparameters in the KRR: 1. α, which controls the strength of regularization 2. γ in the RBF:

$$\text{rbf}(x_i, x_j) = \exp(-\gamma(x_i - x_j)^2),$$

or

$$\gamma = \frac{1}{2\sigma^2},$$

where σ is the width of the Gaussian function: $G(x_i) = \exp\left(\frac{-(x_i-\mu)^2}{2\sigma^2}\right)$.

When the hyperparameters are tuned, gammas of different orders of magnitude are looped through. `GridSearchCV` is then used to find out the alpha of different orders of magnitude which produces the highest r square value for a given gamma. `GridSearchCV` is called two times to refine the hyperparameters. In the second round of tuning, the range of alpha and gamma is limited to 0.5~5*optimal value in the first round.

6.2 Regression using unscaled data

```
In [13]: 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)
         alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])
         parameter_ranges = {'alpha':alphas}
```

```
        KRR_r2_test_list = []
        KRR_coef_list = []
        KRR_gamma_list = []
        KRR_alpha_list = []

        for gamma in gammas:
            KRR = KernelRidge(kernel='rbf',gamma=gamma)
            KRR_search = GridSearchCV(KRR, parameter_ranges, cv=kf)
            KRR_search.fit(X_train_test, y_train_test)
            KRR_r2_test_list.append(KRR_search.best_score_)
            KRR_gamma_list.append(KRR_search.best_estimator_.gamma)
            KRR_alpha_list.append(KRR_search.best_estimator_.alpha)
            KRR_coef_list.append(KRR_search.best_estimator_.dual_coef_)

        best_i = np.array(KRR_r2_test_list).argmax()
        KRR_r2_best_1= KRR_r2_test_list[best_i]
        KRR_gamma_best_1 = KRR_gamma_list[best_i]
        KRR_alpha_best_1 = KRR_alpha_list[best_i]
        KRR_coef_best_1 = KRR_coef_list[best_i]

        print("Highest r2={:.3f},\toptimal gamma={:.0e},\toptimal alpha={:.0e}".format(KRR_r2_t
```

```
Highest r2=0.963,       optimal gamma=1e-04,        optimal alpha=1e-03
```

In the first round of the hyperparameter tuning, r2 score is high, indicating good performance of the model.

```
In [14]: #Second interation with smaller range of alphas and gammas
        gammas = np.array([KRR_gamma_best_1*0.5,KRR_gamma_best_1*0.75,KRR_gamma_best_1,KRR_gamm
        alphas = np.array([KRR_alpha_best_1*0.5,KRR_alpha_best_1*0.75,KRR_alpha_best_1,KRR_alph
        parameter_ranges = {'alpha':alphas}

        KRR_r2_test_list = []
        KRR_coef_list = []
        KRR_gamma_list = []
        KRR_alpha_list = []

        for gamma in gammas:
            KRR = KernelRidge(kernel='rbf',gamma=gamma)
            KRR_search = GridSearchCV(KRR, parameter_ranges, cv=kf)
            KRR_search.fit(X_train_test, y_train_test)
            KRR_r2_test_list.append(KRR_search.best_score_)
            KRR_gamma_list.append(KRR_search.best_estimator_.gamma)
            KRR_alpha_list.append(KRR_search.best_estimator_.alpha)
            KRR_coef_list.append(KRR_search.best_estimator_.dual_coef_)

        best_i = np.array(KRR_r2_test_list).argmax()
```

```
        KRR_r2_best_2= KRR_r2_test_list[best_i]
        KRR_gamma_best = KRR_gamma_list[best_i]
        KRR_alpha_best = KRR_alpha_list[best_i]
        KRR_coef_best = KRR_coef_list[best_i]

        print("Highest r2={:.3f},\toptimal gamma={:.0e},\toptimal alpha={:.0e}".format(KRR_r2_t
```

```
Highest r2=0.969,       optimal gamma=7e-05,        optimal alpha=8e-04
```

In the second round of the hyperparameter tuning, the optimal values of gamma and alpha are different. We don't see a significant improvement in r2 scores, but in the first round the models are already well-performed.

6.3 Regression using scaled data with hyperparameter tuning

```
In [16]: #Regression with scaled data
         #GridSearchCV alpha iterate gamma

         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)
         alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])
         parameter_ranges = {'alpha':alphas}

         KRR_r2_test_scaled_list = []
         KRR_coef_scaled_list = []
         KRR_gamma_scaled_list = []
         KRR_alpha_scaled_list = []

         for gamma in gammas:
             KRR = KernelRidge(kernel='rbf',gamma=gamma)
             KRR_search = GridSearchCV(KRR, parameter_ranges, cv=kf)
             KRR_search.fit(X_train_test_scaled, y_train_test)
             KRR_r2_test_scaled_list.append(KRR_search.best_score_)
             KRR_gamma_scaled_list.append(KRR_search.best_estimator_.gamma)
             KRR_alpha_scaled_list.append(KRR_search.best_estimator_.alpha)
             KRR_coef_scaled_list.append(KRR_search.best_estimator_.dual_coef_)


         best_i = np.array(KRR_r2_test_scaled_list).argmax()
         KRR_r2_best_scaled1= KRR_r2_test_scaled_list[best_i]
         KRR_gamma_best_scaled_1 = KRR_gamma_scaled_list[best_i]
         KRR_alpha_best_scaled_1 = KRR_alpha_scaled_list[best_i]
         KRR_coef_best_scaled_1 = KRR_coef_scaled_list[best_i]

         print("Highest r2={:.3f},\toptimal gamma={:.0e},\toptimal alpha={:.0e}".format(KRR_r2_t
```

```
Highest r2=0.987,       optimal gamma=5e-03,        optimal alpha=1e-03
```

By using scaled data for regression, the r square score is able to improve compared with the model using unscaled data.

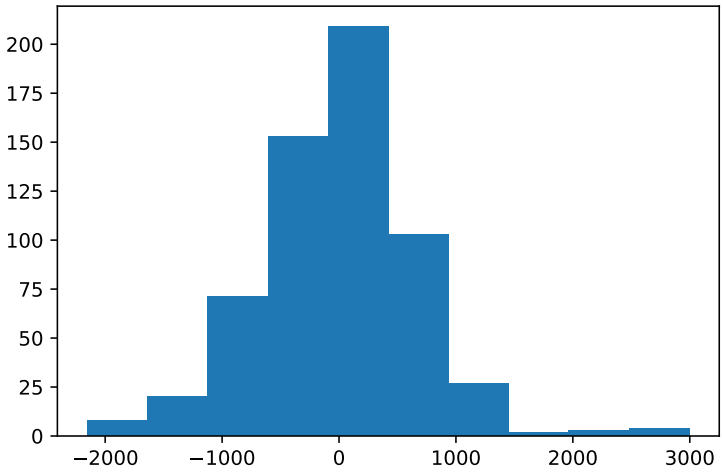
```
In [17]: #second interation with smaller range of alphas and gammas

         gammas = np.array([KRR_gamma_best_scaled_1*0.5,KRR_gamma_best_scaled_1*0.75,KRR_gamma_b
         alphas = np.array([KRR_alpha_best_scaled_1*0.5,KRR_alpha_best_scaled_1*0.75,KRR_alpha_b
         parameter_ranges = {'alpha':alphas}

         KRR_r2_test_scaled_list = []
         KRR_coef_scaled_list = []
         KRR_gamma_scaled_list = []
         KRR_alpha_scaled_list = []

         for gamma in gammas:
             KRR = KernelRidge(kernel='rbf',gamma=gamma)
             KRR_search = GridSearchCV(KRR, parameter_ranges, cv=kf)
             KRR_search.fit(X_train_test_scaled, y_train_test)
             KRR_r2_test_scaled_list.append(KRR_search.best_score_)
             KRR_gamma_scaled_list.append(KRR_search.best_estimator_.gamma)
             KRR_alpha_scaled_list.append(KRR_search.best_estimator_.alpha)
             KRR_coef_scaled_list.append(KRR_search.best_estimator_.dual_coef_)


         best_i = np.array(KRR_r2_test_scaled_list).argmax()
         KRR_r2_best_scaled2= KRR_r2_test_scaled_list[best_i]
         KRR_gamma_best_scaled = KRR_gamma_scaled_list[best_i]
         KRR_alpha_best_scaled = KRR_alpha_scaled_list[best_i]
         KRR_coef_best_scaled = KRR_coef_scaled_list[best_i]

         print("Highest r2={:.3f},\toptimal gamma={:.0e},\toptimal alpha={:.0e}".format(KRR_r2_t

Highest r2=0.987,        optimal gamma=5e-03,        optimal alpha=3e-03
```

After the search range of the hyperparameters is refined, the optimal values change a little bit, but the r^2 value remains nearly the same.

To avoid overfitting, smoother models are preferred. We look at how using different input data (unscaled and scaled) affects the size and distribution of the coefficients in KRR models. Histograms are drawn and the largest absolute values of the coefficients in KRR using unscaled and scaled data are printed below.

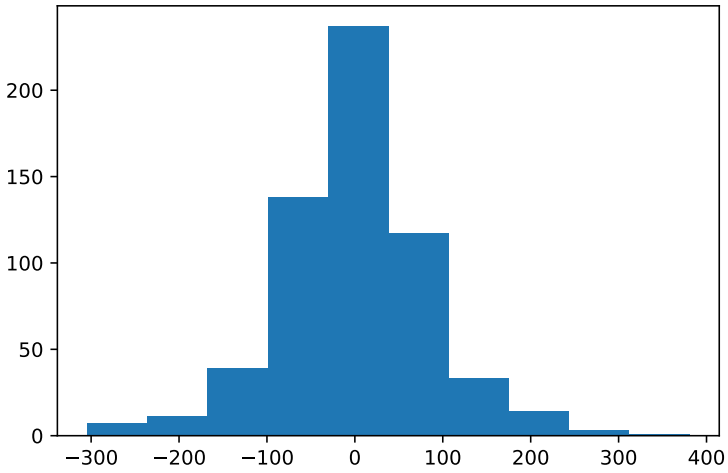
```
In [19]: print('The model using unscaled data has {} coefficients.'.format(len(KRR_coef_best)))

         fig, ax = plt.subplots()
         ax.hist(KRR_coef_best)
         print('The largest coefficient is {:.3f}.'.format(max(abs(KRR_coef_best))[0]));
         KRR_largest_coef=max(abs(KRR_coef_best))[0]
```

```
The model using unscaled data has 600 coefficients.
The largest coefficient is 2994.760.
```




```
In [20]: print('The model using scaled data has {} coefficients.'.format(len(KRR_coef_best_scale

         fig, ax = plt.subplots()
         ax.hist(KRR_coef_best_scaled)
         print('The largest coefficient is {:.3f}.'.format(max(abs(KRR_coef_best_scaled))[0]));
         KRR_largest_coef_scaled=max(abs(KRR_coef_best_scaled))[0]
```

```
The model using scaled data has 600 coefficients.
The largest coefficient is 380.267.
```



The coefficients are normally distributed for two models. The values of coefficients in KRR models, which include regularization terms in the loss function, are in a reasonable range. One thing to notice is that the size of the coefficients is one order of magnitude smaller after scaling the data by comparing the largest coefficient in the model. This indicates that KRR using scaled data is more likely to be not overfitted with high accuracy.

7 LASSO regression

Lasso regression is a type of linear regression that uses shrinkage. LASSO is the abbreviation of Least Absolute Shrinkage and Selection Operator. This particular type of regression is well-suited for models showing high levels of muticollinearity or to automate certain parts of model selection, such as parameter elimination. As we have been taught during the lecture, the loss function for LASSO is defined as:

$$L_{\text{LASSO}} = \sum_i {\varepsilon_i}^2 + \alpha ||\vec{w}||_1$$

Despite the fact that our multi-linear model works very well, overfitting may still be a big issue. Parameter regularization is therefore an effectively way to avoid such complexity.

Herein, we introduce Lasso Regression based on the follow reasons: 1. Overfitting issue from our baseline model should be resolved. 2. Dropping some unimportant features and simplify our model.

Because our multi-linear model has very good performance, the first part is Lasso regularization combined with multi-linear model which has 35 features. The second part of Lasso regularization has different input, which is rbf kernel matrix, to see if Lasso regression improves the poor performance of rbf kernel in our baseline model (even though very unlikely).

7.1 LASSO multi-linear regression

7.1.1 Regression using unscaled data

```
In [21]: alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])

         parameter_set = {'alpha':alphas}

         LASSO = Lasso(tol=1)
         LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
         LASSO_search.fit(X_train_test, y_train_test)


         best_alpha = LASSO_search.best_estimator_.alpha
         r_2 = LASSO_search.best_score_
         coeffs = LASSO_search.best_estimator_.coef_
         nonzero = [f for f in np.isclose(coeffs,0) if f == True]
         num_dropped = len(nonzero)

         r2s_alpha = LASSO_search.cv_results_['mean_test_score']
```

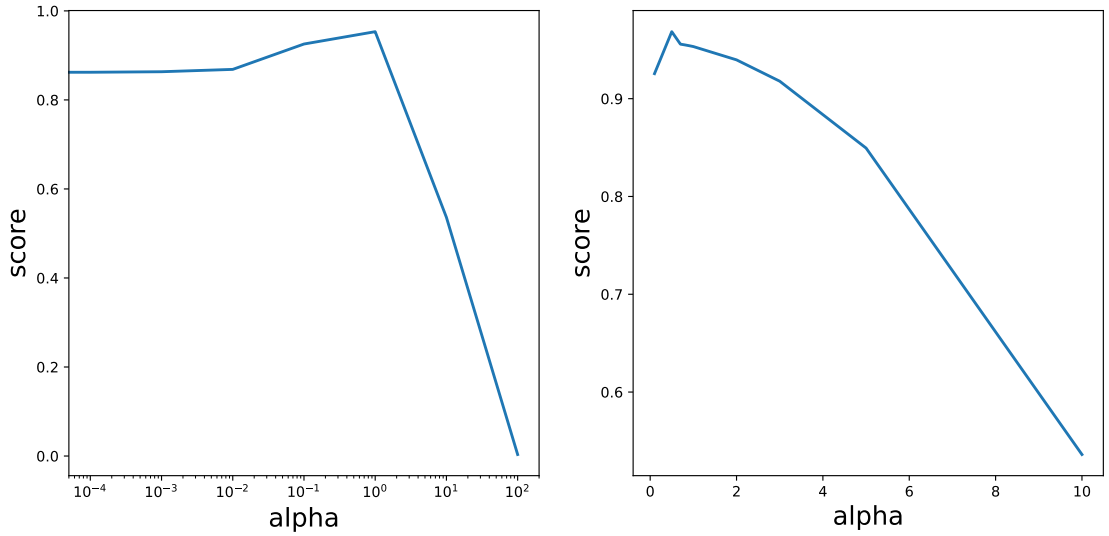
```
        print('Best alpha is: {}'.format(best_alpha))
        print('r square is: {}'.format(r_2))
        print('Number of dropped features: {}'.format(num_dropped))

        #Second refinement
        print('\n====== Second GridSearchCV Refinement ======\n')

        alphas_fine = np.array([0.1, 0.5, 0.7, 0.8, 0.9, 1, 2, 3, 5, 10])

        parameter_set = {'alpha':alphas_fine}

        LASSO = Lasso(tol=1)
        LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
        LASSO_search.fit(X_train_test, y_train_test)

        best_alpha = LASSO_search.best_estimator_.alpha
        r_2 = LASSO_search.best_score_
        coeffs = LASSO_search.best_estimator_.coef_
        nonzero = [f for f in np.isclose(coeffs,0) if f == True]
        num_dropped = len(nonzero)

        r2s_alpha_fine = LASSO_search.cv_results_['mean_test_score']
        print(r2s_alpha_fine)

        print('Best alpha is: {}'.format(best_alpha))
        print('r square is: {}'.format(r_2))
        print('Number of dropped features: {}'.format(num_dropped))
```

```
Best alpha is: 1.0
r square is: 0.9532856275794167
Number of dropped features: 28

====== Second GridSearchCV Refinement ======

[0.92556555 0.96850343 0.95574649 0.95500959 0.9541893  0.95328563
 0.93968123 0.91786376 0.84949036 0.53612941]
Best alpha is: 0.5
r square is: 0.9685034288939386
Number of dropped features: 29
```

```
In [22]: fig, axes = plt.subplots(1, 2, figsize=(13, 6))

        axes[0].plot(alphas, r2s_alpha, alpha=1)
        axes[0].set_xlabel('alpha')
        axes[0].set_ylabel('score')
        axes[0].set_xscale('log')
        axes[0].set_xlim(5e-5, 2e2)
```

```
axes[1].plot(alphas_fine, r2s_alpha_fine, alpha=1)
axes[1].set_xlabel('alpha')
axes[1].set_ylabel('score')
```

Out[22]: Text(0, 0.5, 'score')



Considering the trade-off between precision and model simplification. The goal of this work is to predict the gas uptate by feature of MOFs and adsorbates with very high accuracy. But from experimental aspect, it is definitely much more convenient if a relatively small amount of descriptors could sufficiently predict gas uptakes since features could be difficult and time-consuming to be obtained experimentally. Consequently, the above r_{square} vs. α figures is plotted to demonstrate alpha optimization.

r^2 vs. α plots (left: broad scan; right: finer scan) show that if α is too large, important features are ignored and thus causes extremely low or even negative r^2. This is reasonable because extreme regularization means that no regression is applied (first term of the loss function neglected). In our case, we want α as large as possible without sacrifices r^2. From our result, optimal α is 0.5.

7.1.2 Regression using scaled data

In [23]:
```
alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])

parameter_set = {'alpha':alphas}

LASSO = Lasso(tol=1)
LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
LASSO_search.fit(X_train_test_scaled, y_train_test)

best_alpha = LASSO_search.best_estimator_.alpha
r_2 = LASSO_search.best_score_
```

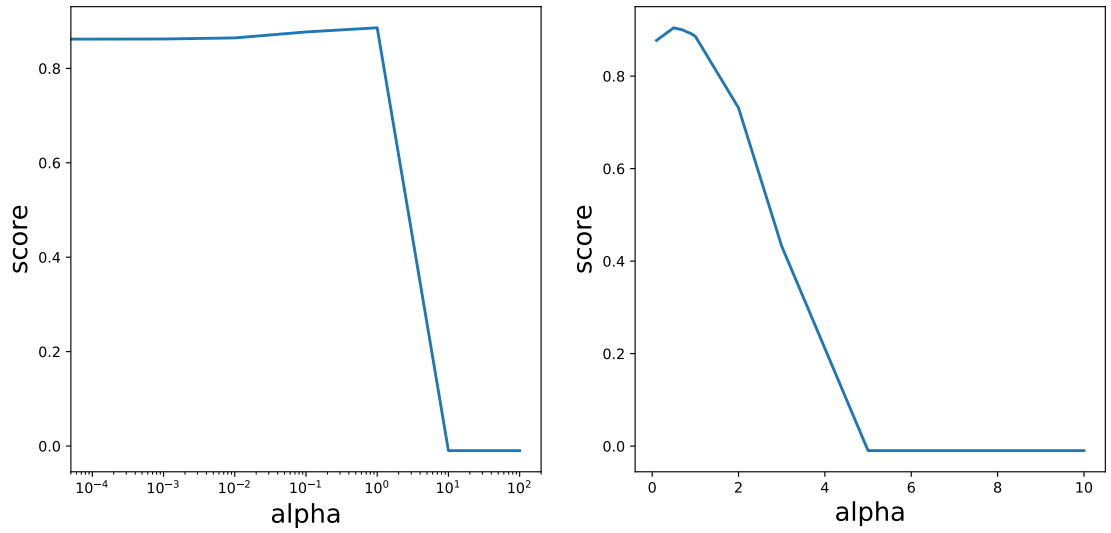
```
        coeffs = LASSO_search.best_estimator_.coef_
        nonzero = [f for f in np.isclose(coeffs,0) if f == True]
        num_dropped = len(nonzero)

        r2s_alpha = LASSO_search.cv_results_['mean_test_score']


        print('Best alpha is: {}'.format(best_alpha))
        print('r square is: {}'.format(r_2))
        print('Number of dropped features: {}'.format(num_dropped))

        #Second refinement
        print('\n====== Second GridSearchCV Refinement ======\n')

        alphas_fine = np.array([0.1, 0.5, 0.7, 0.8, 0.9, 1, 2, 3, 5, 10])

        parameter_set = {'alpha':alphas_fine}

        LASSO = Lasso(tol=1)
        LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
        LASSO_search.fit(X_train_test_scaled, y_train_test)

        best_alpha = LASSO_search.best_estimator_.alpha
        r_2 = LASSO_search.best_score_
        coeffs = LASSO_search.best_estimator_.coef_
        nonzero = [f for f in np.isclose(coeffs,0) if f == True]
        num_dropped = len(nonzero)

        r2s_alpha_fine = LASSO_search.cv_results_['mean_test_score']

        print('Best alpha is: {}'.format(best_alpha))
        print('r square is: {}'.format(r_2))
        print('Number of dropped features: {}'.format(num_dropped))
        print(coeffs)
```

```
Best alpha is: 1.0
r square is: 0.8859648318747583
Number of dropped features: 33

====== Second GridSearchCV Refinement ======

Best alpha is: 0.5
r square is: 0.9044343072962865
Number of dropped features: 32
[ 1.57286766 -0.          0.         -0.          0.         -0.
 -0.         -0.          0.         -0.          0.          0.
  0.          0.          0.         -0.          0.         -0.
 -0.         -0.         -0.         -0.          0.         -0.
```

```
  0.          0.         -0.          0.         -0.         -0.
  0.23595851  0.          0.          0.          2.17996085]
```

```
In [24]: fig, axes = plt.subplots(1, 2, figsize=(13, 6))

         axes[0].plot(alphas, r2s_alpha, alpha=1)
         axes[0].set_xlabel('alpha')
         axes[0].set_ylabel('score')
         axes[0].set_xscale('log')
         axes[0].set_xlim(5e-5, 2e2)

         axes[1].plot(alphas_fine, r2s_alpha_fine, alpha=1)
         axes[1].set_xlabel('alpha')
         axes[1].set_ylabel('score')
```

```
Out[24]: Text(0, 0.5, 'score')
```



After rescaling X matrix across data points, surprisingly, the performance of the model became slightly worse. But generally, multi-linear Lasso regression is a good approach to shrink and largely simplify descriptors the with high accuracy (32 features are dropped based on the result, only two features are really important). This is a very exciting result that we can only use 3 most important features to describe gas uptakes for most of the cases!

The 3 most important features are: - V_f, accessible pore volume per unit cell of MOF; - P_c, critical pressure (bar) of absorbate (related to the equation of state of real gas), and - gas uptake obtained from the rigid structure.

This makes sense as they come from three separate data sets.

We may further explore potential benefits of feature engineering in the final report.

7.2 LASSO Regression with RBF Kernel

7.2.1 Regression using unscaled data

In [25]:
```
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, 4))

alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])
parameter_set = {'alpha':alphas}


for j, gamma in enumerate(gammas):

    X_train_kernel = rbf_kernel(X_train_test, X_train_test, gamma = gamma)

    LASSO = Lasso(tol=1)
    LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
    LASSO_search.fit(X_train_kernel, y_train_test)

    r2_matrix[j, 0] = gamma
    r2_matrix[j, 1] = LASSO_search.best_estimator_.alpha
    r2_matrix[j, 2] = LASSO_search.best_score_
    coeffs = LASSO_search.best_estimator_.coef_
    nonzero = [f for f in np.isclose(coeffs,0) if f == True]
    num_dropped = len(nonzero)
    r2_matrix[j, -1] = int(num_dropped)

n = r2_matrix[:, 2].argmax()
# print(r2_matrix)

print('Best gamma is: {:.0e}'.format(r2_matrix[n,0]))
print('Best alpha is: {:.0e}'.format(r2_matrix[n,1]))
print('r square is: {:.3f}'.format(r2_matrix[n,2]))
print('Number of dropped features: {}'.format(r2_matrix[n,-1]))


#Second refinement
print('\n====== Second GridSearchCV Refinement ======\n')

sigmas = np.array([5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
gammas = 1./(2*sigmas**2)
r2_matrix = np.zeros((gammas.size, 4))

alphas = np.array([1e-4, 5e-4, 8e-4, 9e-4, 1e-3, 2e-3, 3e-3, 5e-3, 7e-3])
parameter_set = {'alpha':alphas}
```

```
        for j, gamma in enumerate(gammas):

            X_train_kernel = rbf_kernel(X_train_test, X_train_test, gamma = gamma)

            LASSO = Lasso(tol=1)
            LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
            LASSO_search.fit(X_train_kernel, y_train_test)

            r2_matrix[j, 0] = gamma
            r2_matrix[j, 1] = LASSO_search.best_estimator_.alpha
            r2_matrix[j, 2] = LASSO_search.best_score_
            coeffs = LASSO_search.best_estimator_.coef_
            nonzero = [f for f in np.isclose(coeffs,0) if f == True]
            num_dropped = len(nonzero)
            r2_matrix[j, -1] = int(num_dropped)

        n = r2_matrix[:, 2].argmax()
        # print(r2_matrix)

        print('Best gamma is: {:.0e}'.format(r2_matrix[n,0]))
        print('Best alpha is: {:.0e}'.format(r2_matrix[n,1]))
        print('r square is: {:.3f}'.format(r2_matrix[n,2]))
        print('Number of dropped features: {}'.format(r2_matrix[n,-1]))
```

```
Best gamma is: 5e-03
Best alpha is: 1e-03
r square is: 0.445
Number of dropped features: 92.0

====== Second GridSearchCV Refinement ======

Best gamma is: 3e-03
Best alpha is: 8e-04
r square is: 0.453
Number of dropped features: 71.0
```

Not surprisingly, unscaled X results in poor r square performances. We want to address here that using rbf kernel matrix seems to be unclever because it averages out each row into a single value (i.e. assuming all features have same contributions). Apparently, ignoring physical meanings of descriptors causes huge problem.

7.2.2 Regression using scaled data

```
In [26]: 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, 4))

         alphas = np.array([1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1, 10, 100])
```

```
parameter_set = {'alpha':alphas}


for j, gamma in enumerate(gammas):

    X_train_kernel = rbf_kernel(X_train_test_scaled, X_train_test_scaled, gamma = gamma

    LASSO = Lasso(tol=1)
    LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
    LASSO_search.fit(X_train_kernel, y_train_test)

    r2_matrix[j, 0] = gamma
    r2_matrix[j, 1] = LASSO_search.best_estimator_.alpha
    r2_matrix[j, 2] = LASSO_search.best_score_
    coeffs = LASSO_search.best_estimator_.coef_
    nonzero = [f for f in np.isclose(coeffs,0) if f == True]
    num_dropped = len(nonzero)
    r2_matrix[j, -1] = num_dropped

n = r2_matrix[:, 2].argmax()
# print(r2_matrix)

print('Best gamma is: {:.0e}'.format(r2_matrix[n,0]))
print('Best alpha is: {:.0e}'.format(r2_matrix[n,1]))
print('r square is: {:.3f}'.format(r2_matrix[n,2]))
print('Number of dropped features: {}'.format(r2_matrix[n,-1]))


#Second refinement
print('\n====== Second GridSearchCV Refinement ======\n')

sigmas = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
gammas = 1./(2*sigmas**2)
r2_matrix = np.zeros((gammas.size, 4))

alphas = np.array([1e-5, 2e-5, 3e-5, 4e-5, 5e-5])
parameter_set = {'alpha':alphas}


for j, gamma in enumerate(gammas):

    X_train_kernel = rbf_kernel(X_train_test_scaled, X_train_test_scaled, gamma = gamma

    LASSO = Lasso(tol=1)
    LASSO_search = GridSearchCV(LASSO, parameter_set, cv =kf)
    LASSO_search.fit(X_train_kernel, y_train_test)

    r2_matrix[j, 0] = gamma
```

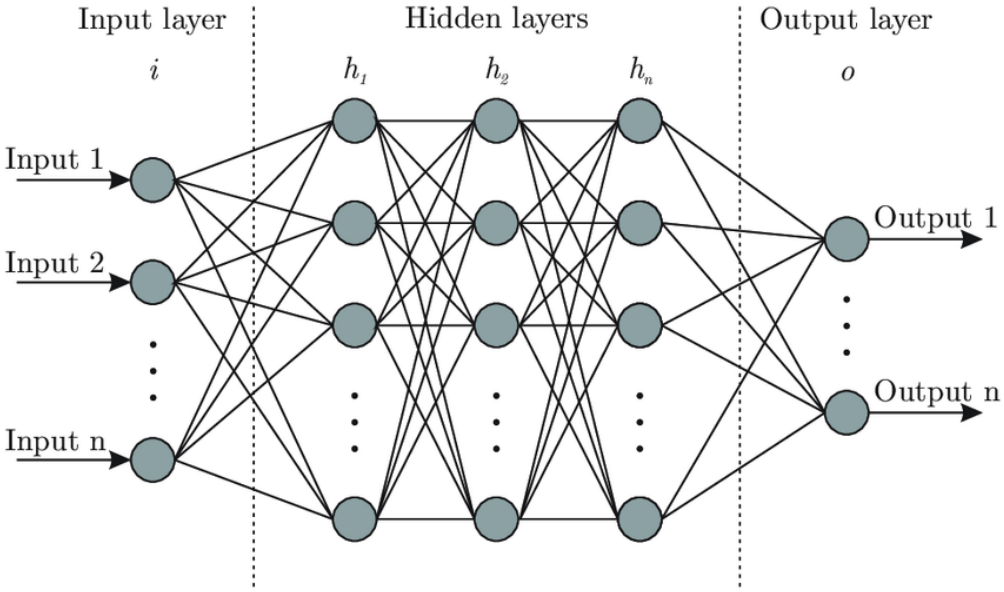
```
            r2_matrix[j, 1] = LASSO_search.best_estimator_.alpha
            r2_matrix[j, 2] = LASSO_search.best_score_
            coeffs = LASSO_search.best_estimator_.coef_
            nonzero = [f for f in np.isclose(coeffs,0) if f == True]
            num_dropped = len(nonzero)
            r2_matrix[j, -1] = num_dropped

        n = r2_matrix[:, 2].argmax()
        # print(r2_matrix)
        # print(coeffs.shape)

        print('Best gamma is: {:.0e}'.format(r2_matrix[n,0]))
        print('Best alpha is: {:.0e}'.format(r2_matrix[n,1]))
        print('r square is: {:.3f}'.format(r2_matrix[n,2]))
        print('Number of dropped features: {}'.format(r2_matrix[n,-1]))
```

```
Best gamma is: 2e-02
Best alpha is: 1e-04
r square is: 0.933
Number of dropped features: 3.0

====== Second GridSearchCV Refinement ======

Best gamma is: 2e-02
Best alpha is: 5e-05
r square is: 0.933
Number of dropped features: 1.0
```

After rescaling x matrix across data points, there is a huge improvement of model performance. However, just 1 of the 600 features is dropped (non-parametric model), and optimal alpha values are all very small, i.e. the model is not regularized at all, parameters not simplified.language

8 MLPRegressor (neural network)

Neural network (NN) is a powerful set of algorithms inspired by human brains. As mentioned in the class, perceptron classifier model can be seen as a single layer NN classifier model, and NN models can be seen as multi-layer perceptron models by applying the logistic or ReLU functions several times on the perceptron model outputs. They are good at nonlinear data processing and are used extensively in many fields, such as image processing, natural language processing and forecasting.



Here we intend to apply one of NN regression models on our project to:

1. learn the mechanism of NN models, which hyperparameters they have, and how to tune them, and
2. briefly compare NN prediction results with other prediction models covered in the class.

The NN model that we use is `MLPRegressor` in `sklearn`, where `MLP` means multi-layer Perceptron. It is capable of processing nonlinear data, but it has a nonconvex loss function, which makes it hard to find the global optimum. It also involves a number of hyperparameters, such as the number of hidden neurons and layers. The focus of this section is to finely tuning the hyperparameters of `MLPRegressor` to try to obtain a NN model with improved prediction ability.

8.1 Hyperparameter tuning strategy

There are three main types of hyperparameters:

2. the number of hidden layers ($L \in \mathbb{N}$),
3. the number of hidden neurons ($n_l \in \mathbb{N}, l \in \{1, 2, ..., L\}$), and
4. loss function types (Logistic function or Rectified Linear Unit (ReLU) function).

$$\mathcal{L}_{\text{logistic}}(x) = \max(0, x)$$

$$\mathcal{L}_{\text{ReLU}}(x) = \ln(1 + e^{-x})$$

The number of parameters in the model grows linearly with n_l and exponentially with L. Therefore, it may be computationally expensive to simultaneously tune n_l and L. An alternative way is to first tune n_1 for a single-layer NN model, then fix n_1 and tune n_2 and so on. So we propose to tune the hyperparameter for the NN model using the alternative way while limiting the upper bound of n_i to 100 and L to 3. The tuning process will be conducted twice with Logistic and ReLu as loss functions respectively.

8.2 Hyperparameter tuning for MLPRegressor with Logistic loss function

8.2.1 Tuning for 1-layer model

First a rough value list of n_1 is searched through:

```
In [74]: layer_list_n1 = [1] + list(range(10, 110, 10))
         print("Initial n1 search space: {}".format(layer_list_n1))
         layer_dict_n1 = {
             'hidden_layer_sizes': layer_list_n1
         }
```

```
Initial n1 search space: [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

```
In [75]: loss_fun = 'logistic'

         model_NN_1 = MLPRegressor(max_iter=500, activation=loss_fun)
         model_NN_1_search = GridSearchCV(model_NN_1, layer_dict_n1, n_jobs=-1, cv=kf)
         model_NN_1_search.fit(X_train_test, y_train_test.ravel())
         model_NN_1_best = model_NN_1_search.best_estimator_
         print("Best model score: {:.3f}".format(model_NN_1_search.best_score_))
         print("Best n1: {:d}".format(model_NN_1_best.hidden_layer_sizes))
```

```
Best model score: 0.954
Best n1: 90
```

```
/Users/chengpengfei/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/multilayer_perc
  % self.max_iter, ConvergenceWarning)
```

Then a more refined search space of n_1 is used:

```
In [76]: layer_list_n1_refined = list(range(max(model_NN_1_best.hidden_layer_sizes - 10, 1), min
         print("Refined n1 search space: {}".format(layer_list_n1_refined))
         layer_dict_n1_refined = {
             'hidden_layer_sizes': layer_list_n1_refined
         }

         model_NN_1_refined = MLPRegressor(max_iter=500, activation=loss_fun)
         model_NN_1_refined_search = GridSearchCV(model_NN_1_refined, layer_dict_n1_refined, n_j
         model_NN_1_refined_search.fit(X_train_test, y_train_test.ravel())
         logistic_NN_1_refined_best = model_NN_1_refined_search.best_estimator_
         logistic_NN_1_refined_score = model_NN_1_refined_search.best_score_
         best_n1 = logistic_NN_1_refined_best.hidden_layer_sizes
         print("Best model score: {:.3f}".format(logistic_NN_1_refined_score))
         print("Best n1: {:d}".format(best_n1))
```

```
Refined n1 search space: [80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
Best model score: 0.959
Best n1: 96
```

8.2.2 Tuning for 2-layer model

Denote n_1^* as the optimal value for the number of neurons for the first layer. A list of 2-tuples is defined as the search space for the 2-layer NN model. The first element is fixed at n_1^*, the second element varies from 1 to 100.

```
In [77]: list_temp = [1] + list(range(10, 110, 10))

         layer_list_n2 = []
         for i in list_temp:
             layer_list_n2.append((best_n1, i))

         print("Initial (n1, n2) search space: {}".format(layer_list_n2))
         layer_dict_n2 = {
             'hidden_layer_sizes': layer_list_n2
         }

         model_NN_2 = MLPRegressor(max_iter=500, activation=loss_fun)
         model_NN_2_search = GridSearchCV(model_NN_2, layer_dict_n2, n_jobs=-1, cv=kf)
         model_NN_2_search.fit(X_train_test, y_train_test.ravel())
         model_NN_2_best = model_NN_2_search.best_estimator_
         print("Best model score: {:.3f}".format(model_NN_2_search.best_score_))
         print("Best (n1, n2): {}".format(model_NN_2_best.hidden_layer_sizes))
```

```
Initial (n1, n2) search space: [(96, 1), (96, 10), (96, 20), (96, 30), (96, 40), (96, 50), (96,
Best model score: 0.961
Best (n1, n2): (96, 70)
```

```
/Users/chengpengfei/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/multilayer_perc
  % self.max_iter, ConvergenceWarning)
```

It is worth noting that (n_1^*, n_2) leads to a slightly worse results than the single (n_1^*). The reason may mainly be the non-convexity of the loss function, which causes the solver to converge to a local optimum and there is no guarantee that a model with more layers following this strategy yields a better solution.

Similarly, a refined space is used for n_2:

```
In [78]: list_temp = list(range(max(model_NN_2_best.hidden_layer_sizes[1] - 10, 1), min(102, mod

         layer_list_n2_refined = []
         for i in list_temp:
             layer_list_n2_refined.append((best_n1, i))

         print("Refined (n1, n2) search space: {}".format(layer_list_n2_refined))
         layer_dict_n2_refined = {
             'hidden_layer_sizes': layer_list_n2_refined
         }
```

```
        model_NN_2_refined = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_2_refined_search = GridSearchCV(model_NN_2_refined, layer_dict_n2_refined, n_j
        model_NN_2_refined_search.fit(X_train_test, y_train_test.ravel())
        logistic_NN_2_refined_best = model_NN_2_refined_search.best_estimator_
        logistic_NN_2_refined_score = model_NN_2_refined_search.best_score_
        best_n2 = logistic_NN_2_refined_best.hidden_layer_sizes
        print("Best model score: {:.3f}".format(logistic_NN_2_refined_score))
        print("Best (n1, n2): {}".format(best_n2))
```

```
Refined (n1, n2) search space: [(96, 60), (96, 62), (96, 64), (96, 66), (96, 68), (96, 70), (96,
Best model score: 0.955
Best (n1, n2): (96, 72)
```

8.2.3 Tuning for 3-layer model

n_3 is tuned in a similar manner:

```
In [79]: # initial search
         list_temp = [1] + list(range(10, 110, 10))

         layer_list_n3 = []
         for i in list_temp:
             layer_list_n3.append((best_n1, best_n2[1], i))

         print("Initial (n1, n2, n3) search space: {}".format(layer_list_n3))
         layer_dict_n3 = {
             'hidden_layer_sizes': layer_list_n3
         }

         model_NN_3 = MLPRegressor(max_iter=500, activation=loss_fun)
         model_NN_3_search = GridSearchCV(model_NN_3, layer_dict_n3, n_jobs=-1, cv=kf)
         model_NN_3_search.fit(X_train_test, y_train_test.ravel())
         model_NN_3_best = model_NN_3_search.best_estimator_
         print("Best model score: {:.3f}".format(model_NN_3_search.best_score_))
         print("Best (n1, n2, n3): {}".format(model_NN_3_best.hidden_layer_sizes))

         # refined search
         list_temp = list(range(model_NN_3_best.hidden_layer_sizes[2] - 10, min(100, model_NN_3_
         layer_list_n3_refined = []
         for i in list_temp:
             layer_list_n3_refined.append((best_n1, best_n2[1], i))

         print("Refined (n1, n2, n3) search space: {}".format(layer_list_n3_refined))
         layer_dict_n3_refined = {
             'hidden_layer_sizes': layer_list_n3_refined
         }
```

```
        model_NN_3_refined = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_3_refined_search = GridSearchCV(model_NN_3_refined, layer_dict_n3_refined, n_j
        model_NN_3_refined_search.fit(X_train_test, y_train_test.ravel())
        logistic_NN_3_refined_best = model_NN_3_refined_search.best_estimator_
        logistic_NN_3_refined_score = model_NN_3_refined_search.best_score_
        print("Best model score: {:.3f}".format(logistic_NN_3_refined_score))
        print("Best (n1, n2, n3): {}".format(logistic_NN_3_refined_best.hidden_layer_sizes))
```

```
Initial (n1, n2, n3) search space: [(96, 72, 1), (96, 72, 10), (96, 72, 20), (96, 72, 30), (96,
```

```
/Users/chengpengfei/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/multilayer_perc
  % self.max_iter, ConvergenceWarning)
```

```
Best model score: 0.956
Best (n1, n2, n3): (96, 72, 70)
Refined (n1, n2, n3) search space: [(96, 72, 60), (96, 72, 62), (96, 72, 64), (96, 72, 66), (96,
Best model score: 0.958
Best (n1, n2, n3): (96, 72, 72)
```

```
/Users/chengpengfei/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/multilayer_perc
  % self.max_iter, ConvergenceWarning)
```

8.3 Hyperparameter tuning for MLPRegressor with ReLU loss function

The tuning process for the NN model using ReLU loss function is similar to the process above, except the loss function is specified as ReLU.

```
In [80]: # 1st layer
        layer_list_n1 = [1] + list(range(10, 110, 10))
        print("Initial n1 search space: {}".format(layer_list_n1))
        layer_dict_n1 = {
            'hidden_layer_sizes': layer_list_n1
        }

        loss_fun = 'relu'

        model_NN_1 = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_1_search = GridSearchCV(model_NN_1, layer_dict_n1, n_jobs=-1, cv=kf)
        model_NN_1_search.fit(X_train_test, y_train_test.ravel())
        model_NN_1_best = model_NN_1_search.best_estimator_
        print("Best model score: {:.3f}".format(model_NN_1_search.best_score_))
        print("Best n1: {:d}".format(model_NN_1_best.hidden_layer_sizes))
```

```
        layer_list_n1_refined = list(range(model_NN_1_best.hidden_layer_sizes - 10, min(100, mo
        print("Refined n1 search space: {}".format(layer_list_n1_refined))
        layer_dict_n1_refined = {
            'hidden_layer_sizes': layer_list_n1_refined
        }

        model_NN_1_refined = MLPRegressor(max_iter=750, activation=loss_fun)
        model_NN_1_refined_search = GridSearchCV(model_NN_1_refined, layer_dict_n1_refined, n_j
        model_NN_1_refined_search.fit(X_train_test, y_train_test.ravel())
        relu_NN_1_refined_best = model_NN_1_refined_search.best_estimator_
        relu_NN_1_refined_score = model_NN_1_refined_search.best_score_
        best_n1 = relu_NN_1_refined_best.hidden_layer_sizes
        print("Best model score: {:.3f}".format(relu_NN_1_refined_score))
        print("Best n1: {:d}".format(best_n1))
```

```
Initial n1 search space: [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Best model score: 0.964
Best n1: 60
Refined n1 search space: [50, 52, 54, 56, 58, 60, 62, 64, 66, 68]
Best model score: 0.964
Best n1: 60
```

```
/Users/chengpengfei/anaconda3/lib/python3.7/site-packages/sklearn/neural_network/multilayer_perc
  % self.max_iter, ConvergenceWarning)
```

```
In [81]: # 2nd layer
        list_temp = [1] + list(range(10, 110, 10))

        layer_list_n2 = []
        for i in list_temp:
            layer_list_n2.append((best_n1, i))

        print("Initial (n1, n2) search space: {}".format(layer_list_n2))
        layer_dict_n2 = {
            'hidden_layer_sizes': layer_list_n2
        }

        model_NN_2 = MLPRegressor(max_iter=750, activation=loss_fun)
        model_NN_2_search = GridSearchCV(model_NN_2, layer_dict_n2, n_jobs=-1, cv=kf)
        model_NN_2_search.fit(X_train_test, y_train_test.ravel())
        model_NN_2_best = model_NN_2_search.best_estimator_
        print("Best model score: {:.3f}".format(model_NN_2_search.best_score_))
        print("Best (n1, n2): {}".format(model_NN_2_best.hidden_layer_sizes))
```

```
        list_temp = list(range(max(1, model_NN_2_best.hidden_layer_sizes[1] - 10), min(100, mod

        layer_list_n2_refined = []
        for i in list_temp:
            layer_list_n2_refined.append((best_n1, i))

        print("Refined (n1, n2) search space: {}".format(layer_list_n2_refined))
        layer_dict_n2_refined = {
            'hidden_layer_sizes': layer_list_n2_refined
        }

        model_NN_2_refined = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_2_refined_search = GridSearchCV(model_NN_2_refined, layer_dict_n2_refined, n_j
        model_NN_2_refined_search.fit(X_train_test, y_train_test.ravel())
        relu_NN_2_refined_best = model_NN_2_refined_search.best_estimator_
        relu_NN_2_refined_score = model_NN_2_refined_search.best_score_
        best_n2 = relu_NN_2_refined_best.hidden_layer_sizes
        print("Best model score: {:.3f}".format(relu_NN_2_refined_score))
        print("Best (n1, n2): {}".format(best_n2))
```

```
Initial (n1, n2) search space: [(60, 1), (60, 10), (60, 20), (60, 30), (60, 40), (60, 50), (60,
Best model score: 0.961
Best (n1, n2): (60, 10)
Refined (n1, n2) search space: [(60, 1), (60, 3), (60, 5), (60, 7), (60, 9), (60, 11), (60, 13),
Best model score: 0.963
Best (n1, n2): (60, 17)
```

```
In [82]: # 3rd layer
        list_temp = [1] + list(range(10, 110, 10))

        layer_list_n3 = []
        for i in list_temp:
            layer_list_n3.append((best_n1, best_n2[1], i))

        print("Initial (n1, n2, n3) search space: {}".format(layer_list_n3))
        layer_dict_n3 = {
            'hidden_layer_sizes': layer_list_n3
        }

        model_NN_3 = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_3_search = GridSearchCV(model_NN_3, layer_dict_n3, n_jobs=-1, cv=kf)
        model_NN_3_search.fit(X_train_test, y_train_test.ravel())
        model_NN_3_best = model_NN_3_search.best_estimator_
        print("Best model score: {:.3f}".format(model_NN_3_search.best_score_))
        print("Best (n1, n2, n3): {}".format(model_NN_3_best.hidden_layer_sizes))

        list_temp = list(range(max(model_NN_3_best.hidden_layer_sizes[2] - 10, 1), min(100, mod
```

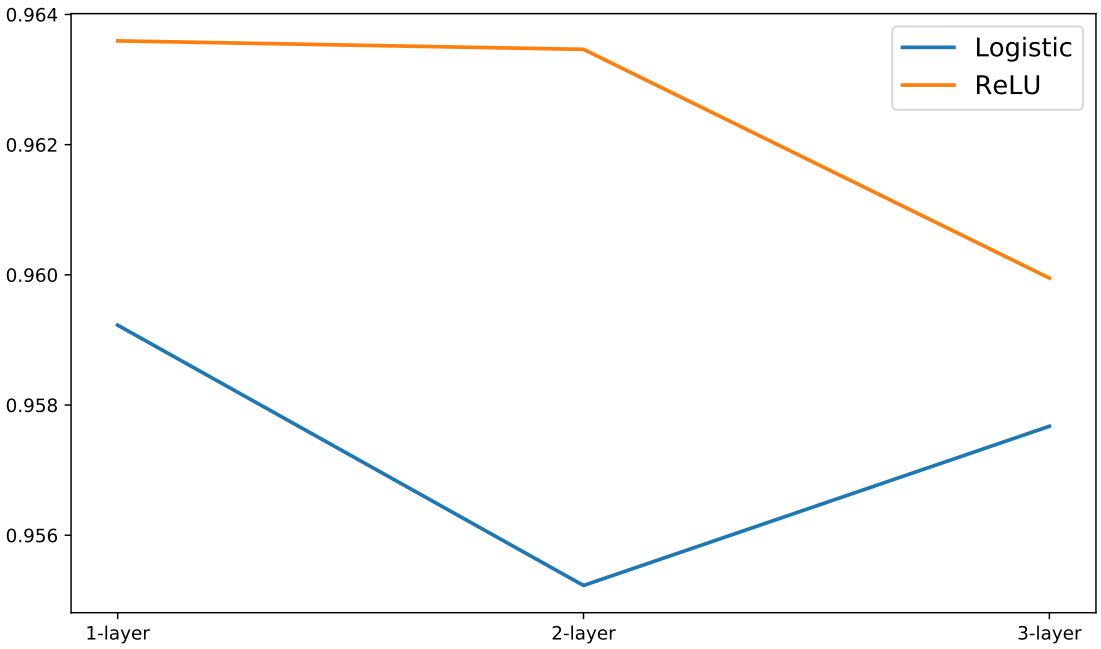
```
        layer_list_n3_refined = []
        for i in list_temp:
            layer_list_n3_refined.append((best_n1, best_n2[1], i))

        print("Refined (n1, n2, n3) search space: {}".format(layer_list_n3_refined))
        layer_dict_n3_refined = {
            'hidden_layer_sizes': layer_list_n3_refined
        }

        model_NN_3_refined = MLPRegressor(max_iter=500, activation=loss_fun)
        model_NN_3_refined_search = GridSearchCV(model_NN_3_refined, layer_dict_n3_refined, n_j
        model_NN_3_refined_search.fit(X_train_test, y_train_test.ravel())
        relu_NN_3_refined_best = model_NN_3_refined_search.best_estimator_
        relu_NN_3_refined_score = model_NN_3_refined_search.best_score_
        print("Best model score: {:.3f}".format(relu_NN_3_refined_score))
        print("Best (n1, n2, n3): {}".format(relu_NN_3_refined_best.hidden_layer_sizes))
```

```
Initial (n1, n2, n3) search space: [(60, 17, 1), (60, 17, 10), (60, 17, 20), (60, 17, 30), (60,
Best model score: 0.961
Best (n1, n2, n3): (60, 17, 40)
Refined (n1, n2, n3) search space: [(60, 17, 30), (60, 17, 32), (60, 17, 34), (60, 17, 36), (60,
Best model score: 0.960
Best (n1, n2, n3): (60, 17, 42)
```

8.4 Discussion

In [83]:
```
x = ['1-layer','2-layer','3-layer']
y_logistic = [logistic_NN_1_refined_score, logistic_NN_2_refined_score, logistic_NN_3_r
y_relu = [relu_NN_1_refined_score, relu_NN_2_refined_score, relu_NN_3_refined_score]

ax, fig = plt.subplots(figsize=(10, 6))
fig.plot(y_logistic, label="Logistic")
fig.plot(y_relu, label="ReLU")
plt.xticks(range(len(x)), x)
fig.legend()
```

Out[83]:
```
<matplotlib.legend.Legend at 0x1a227cc1d0>
```



The r^2 score profiles along with the number of hidden layers are given above for both Logistic and ReLU loss functions. Interestingly, all models obtain high r^2 values, and the increase of the number of hidden layers does not necessarily improve the model performance. There may be several reasons:

1. The 1-layer model is already good enough, and adding extra layers cannot improve its performance, but may introduce some numeric errors causing slight decrease in the r^2 score.

2. The proposed hyperparameter tuning strategy may not be valid. In other words, for a 2-layer model, the best number of neurons for the first layer may not correspond to the best number of neurons of a 1-layer model; there may exist a 2-layer model outperforming the one obtained from the proposed strategy.

Nevertheless, this section shows the ability of NN models for accurate prediction.

Also it is worth noting that training NN models is significantly computationally expensive than KRR and LASSO. We may consider training time as a criterion when comparing all models in the final report.

9 Conclusions and future steps

In the improved model report, we update the validation strategy to improve its robustness, and implement three new regression models: KRR, LASSO and MLPRegressor (neural network). Section 5 clearly shows that the 5-fold validation strategy significantly helps improve model performance through training. Section 6 shows that the addition of regularization can significantly make the model more robust and stable, and may further improve the performance of the RBF kernel regression model. Section 7 shows that the model may be reduced to three features while still keeping good performance. Section 8 shows the difficulty of applying valid and efficient hyperparameter tuning for neural networks, though the model performs well for 1-layer, 2-layer and 3-layer settings.

We plan to carefully analyze the models discussed above and covered the baseline model report, and conduct an overall comparison of all models using the validation strategy in terms of prediction performance, model complexity, training cost, etc. Inspired by the results presented above, we may also conduct other analysis such as principal component analysis to see if feature engineering is crucial to the models, which is not in the original plan.

10 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.