Contents

1. Introduction

The following blocks of code, provide a thorough overview of the data analysis, which were applied to performance and DFT dataset to create a model for Y_MeOH.

These include all of the analysis were carried out to generate the results in the manuscript.

There are 5 main sections:

```
section-1: comparison of preprocessing methods
section-2: determination of optimum model complexity
section-3: generation of SISSO model with optimum complexity
section-4: subgroup discovery (SGD)
section-5: overlay of SISSO prediction, measured values and SGD
result
```

Below each section, respective description is provided.

We begin by importing required libraries, and loading the dataset.

In [4]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from compressed_sensing.sisso import SissoRegressor
import SISSO_Tools as ST
import time
import statistics as st
from sklearn.metrics import r2_score
import shutil
import SGD_functions as SGD
import json
from sklearn.model_selection import KFold
```

In [5]:
```
df = pd.read_excel('Eads_Evac_Performance_Notebook_version.xlsx')
```

Before starting the analysis, the target descriptor (which is to be modelled) and the DFT descriptors are defined for model generation. The used DFT descriptors are selected after removing those descriptors which showed high correlation, i.e. Pearson coefficient above 0.9.

In [6]:
```
# define target property
target_property = 'Y_MeOH'

# determine DFT descriptors
# these descriptors are selected after excluding those which were highly correlate
features = [
  'CO2_ClC([O-])=O_25.0_290', 'CO2_ClC([O-])=O_29.0_290',
  'CO2_ClC([O-])=O_25.0_250',  'CO_S1SC(=O)1_25.0_Ovac0',
  'CO_S1SC(=O)1_29.0_Ovac0',  'CO_Cl[C+][O-]_29.0_250',
  'CHO2_S1SC(=O)[O+]1[H+]_25.0_Ovac0',  'CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac0',
  'CO2_ClO[C+](=O)_25.0_290',  'CO2_ClO[C+](=O)_29.0_290',
  'CO2_ClO[C+](=O)_25.0_250',  'CO2_ClO[C+](=O)_29.0_250',
  'CHO2_S1S[O+]1C(=O)_25.0_Ovac0',  'CHO2_S1S[O+]1C(=O)_29.0_Ovac0',
  'CH2O2_S1SOCO1_25.0_Ovac0',  'CH2O_ClO[CH2-]_25.0_290',
  'CH2O_ClO[CH2-]_29.0_290',  'CH2O_ClO[CH2-]_29.0_250',
  'CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac1',  'CO+O_Cl[C+][O-]_29.0_290',
  'CO+O_Cl[C+][O-]_25.0_250',  'CO+O_Cl[C+][O-]_29.0_250',
  'CH3O+O_Cl[O+](C)_29.0_290',  'CO+OH_Cl[C+][O-]_29.0_290',
  'CO+OH_Cl[C+][O-]_25.0_250',    'Evac_25',    'Evac_29'
  ]
```

2. Comparison_of_preprocessing_methods

The predictive power of regression models are sensitive to data scaling, aka preprocessing. Depending on chosen preprocesing method, the model accuracy might change.

Here, we apply 4 different preprocessing methods to all selected data features. Before comparing their effect, we categorize the data based on Y_MeOH into 3 classes of low, middle, high yield. By doing this, for each of the applied preprocessing methods, 3 SISSO models will be generated, where each time the model is trained with 2 of the performance groups, and tested with the third one.

In [7]:
```
df_1 = df.copy()

# make groups based on performance for CV
min_Y, max_Y = min(df_1['Y_MeOH']), max(df_1['Y_MeOH'])

# assign group ID based on target value
group_ID = []
for val in df_1['Y_MeOH']:
    if val<min_Y+max_Y/3:
        group_ID.append('low_yield')
    elif val>=min_Y+max_Y/3 and val<min_Y+2*max_Y/3:
        group_ID.append('middle_yield')
    else:
        group_ID.append('high_yield')

df_1['yield_group'] = group_ID
```

To start the preprocessing of data features, we shift all of the DFT-obtained descriptors by (Emin - 0.1), where Emin is the lowest obtained Emin across all DFT values. Therefore, all of the values will be above zero, and the smallest value will be set to 0.1.

For more details on preprocessing, refer to the manuscript's main text.

In [8]:
```
# shift Eads
primary_features =['temp']
Eshift = min([min(df[f].values) for f in features]) - 0.1
for feat in features:
    primary_features.append(f"{feat}_Sh")
    df_1[f"{feat}_Sh"] = [val-Eshift for val in df[feat]]

# normalize between 0 and 1 (preprocessing method-i)
primary_features_Nmax = []
for feat in primary_features:
    primary_features_Nmax.append(f"{feat}_Nmax")
    df_1[f"{feat}_Nmax"] = [val/max(df_1[feat]) for val in df_1[feat]]

# normalize by dividing to min (preprocessing method-ii)
primary_features_Nmin = []
for feat in primary_features:
    primary_features_Nmin.append(f"{feat}_Nmin")
    df_1[f"{feat}_Nmin"] = [val/min(df_1[feat]) for val in df_1[feat]]

# add log(T) (preprocessing method-iii)
df_1['log(T)'] = [np.log(t+273) for t in df_1['temp']]
primary_features_log =['log(T)']
for feat in features:
    primary_features_log.append(f"{feat}_Sh")

# divide Eads by T/T0 (preprocessing method-iv)
T0 = 225+273
df_1['T/T0'] = [(t+273)/T0 for t in df_1['temp']]
primary_features_overT = []
for feat in primary_features[1:]:
    primary_features_overT.append(f"{feat}_overT")
    df_1[f"{feat}_overT"] = df_1[feat]/df_1['T/T0']
```

To start the model generation, we create an empty dictionary for saving the RMSE values of each generated model

In [9]:
```
# dictionary for saving CV results
RMSE_list = {'train_RMSE':[], 'test_RMSE':[], 'R2_train':[],'R2_test':[],
             'dim':[],'#op':[],'n_subset':[],'feat_per_it':[],
             '#features':[],'group':[], 'model_type':[]}

# model parameters
# 3D-30 is used as reference
op, dim = 3, 3
feat_per_it = 40
subset = 170


# select primary features
methods = {'Nmax':primary_features_Nmax, 'Nmin':primary_features_Nmin,
           'log_T':primary_features_log, 'without_T':primary_features_overT}
```

For each of the preprocessing methods, 3 SISSO models are generated.

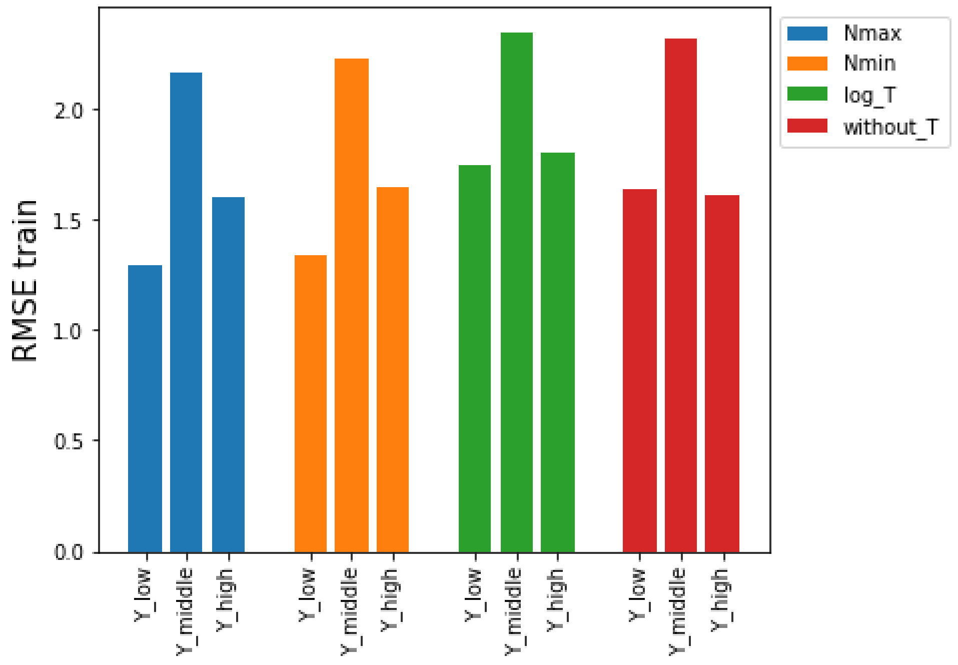
It should be noted that here, all of the features are preprocessed prior to spliting the data into train-test. It is recommended to scale each of train and test set separately, using the same method. (for more details, refer to: https://scikit-learn.org/stable/common_pitfalls.html)

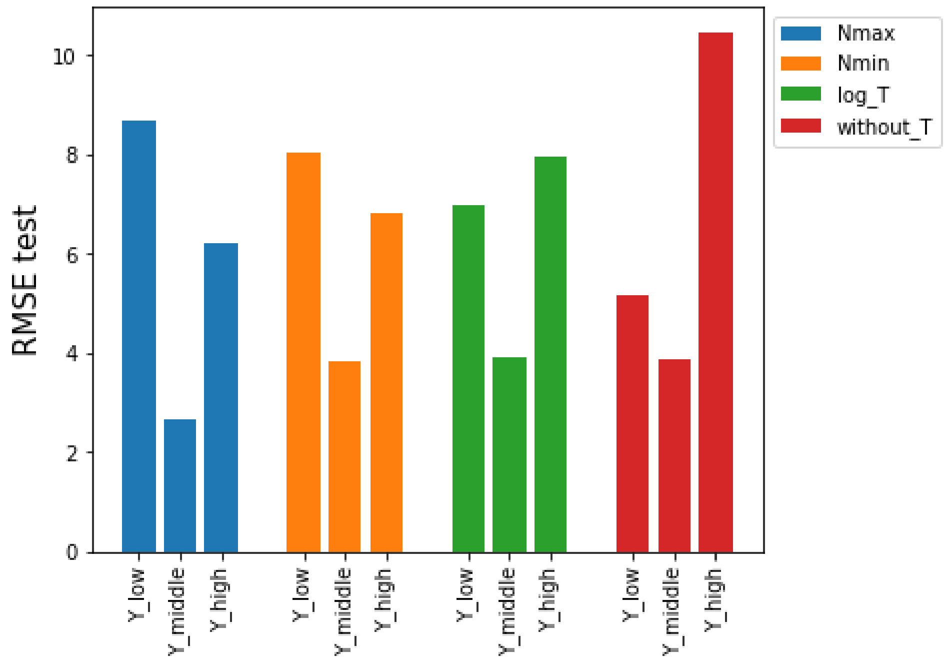
In [13]:
```
# Since run-time for the following code is ~15min+, it is commented out. You can un

'''
for method in methods:
    model_type = method
    selected_primary_features = methods[method]
    # split train and test for CV
    for group in list(df_1['yield_group'].unique()):
        df_train = df_1[df_1['yield_group']!=group]
        df_test = df_1[df_1['yield_group']==group]
        # prepare data
        model_features = selected_primary_features.copy()
        model_features.append(target_property)
        prepared_train = df_train[model_features]
        prepared_test = df_test[model_features]
        # create unit list for all primary features
        # the unit list holds the unit for each of the features in prepared_data
        unit_list = ['C']
        for i in range(prepared_train.shape[1]-1):
            unit_list.append('eV')
    # create SISSO model for training set
        begin = time.time()

        sisso_model, rmse, features_list  = ST.SISSO(prepared_train, target_propert
                                        ['+','-','^2','^0.5','exp','/','*'
                       n_operations=op, dim=dim, features_per_iter=feat_per_it,
                       n_best_subs=subset)
        # predict values
        train_predict = list(ST.evaluate_sisso(sisso_model, prepared_train, feature
        train_measure = list(prepared_train[target_property])
        test_predict = list(ST.evaluate_sisso(sisso_model, prepared_test, features_
        test_measure = list(prepared_test[target_property])

        # get RMSE test
        error = [pred - meas for (pred,meas) in zip(test_predict, test_measure)]
        error_2 = [er**2 for er in error]
        RMSE_test = (sum(error_2)/len(error_2))**0.5

        # save results to dictionary
        RMSE_list['train_RMSE'].append(round(rmse,2))
        RMSE_list['test_RMSE'].append(round(RMSE_test,2))
        RMSE_list['R2_train'].append(r2_score(train_measure,train_predict))
        RMSE_list['R2_test'].append(r2_score(test_measure,test_predict))
        RMSE_list['dim'].append(dim)
        RMSE_list['#op'].append(op)
        RMSE_list['n_subset'].append(subset)
        RMSE_list['feat_per_it'].append(feat_per_it)
        RMSE_list['#features'].append(len(features_list))
        RMSE_list['group'].append(group)
        RMSE_list['model_type'].append(model_type)

# save results
RMSE_df = pd.DataFrame(RMSE_list)
'''

# Since the code is not run, we upload the results
RMSE_df = pd.read_excel('RMSE_df.xlsx')
```

The RMSE for each preprocessing method is plotted below. As it can be seen, the preprocessing method labeled as "Nmax" gives the lowest RMSE, which is then selected for model generation.

In [14]:
```
# plot results for training set
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1,1,figsize=(6,5))
for i,method in enumerate(RMSE_df['model_type'].unique()):
    temp = RMSE_df[RMSE_df['model_type']==method]
    plt.bar([4*i+1,4*i+2,4*i+3], temp['train_RMSE'], label=method)
plt.xticks([1,2,3,5,6,7,9,10,11,13,14,15],4*['Y_low','Y_middle','Y_high'],rotation
plt.legend(bbox_to_anchor=(1,1))
plt.ylabel('RMSE train',size=15)

# plot results for test set
fig, ax = plt.subplots(1,1,figsize=(6,5))
for i,method in enumerate(RMSE_df['model_type'].unique()):
    temp = RMSE_df[RMSE_df['model_type']==method]
    plt.bar([4*i+1,4*i+2,4*i+3], temp['test_RMSE'], label=method)
plt.xticks([1,2,3,5,6,7,9,10,11,13,14,15],4*['Y_low','Y_middle','Y_high'],rotation
plt.legend(bbox_to_anchor=(1,1))
plt.ylabel('RMSE test',size=15)
```

Out[14]:
```
Text(0, 0.5, 'RMSE test')
```





3. Determination_of_optimum_model_complexi

Once the preprocessing method is determined, the model complexity should be evaluated.

For creation of SISSO model, two main parameters are considered: D (dimension) = number of non-zero coefficients, and O (operations) = number of allowed mathematical operations to create each coefficient.

Here, we consider for each parameter 4 levels of complexity, giving a total of 16 complexity levels: from 1D-1O to 4D-4O. To ensure that the high accuracy of a complex model does not lead to overfitting, and thereby losing the power to predict performance of unseen data, we apply a leave-one-group-out cross validation.

To achieve this, for each of the 16 complexity levels, we train 14 different SISSO models: one per each catalyst. That means, each model is trained by performance data from 13 catalyst, and is tested by unseen data from the 14th catalyst. Afterwards, the average RMSE of tested models for unseen data are compared to see which model gives an acceptable accuracy, yet a low RMSE.

Since the run-time for each catalyst is relatively high, we have commented out the script to perform this analysis, and included the final result file. You can get the same results, by running the following block of code.

Here we only show the final results.

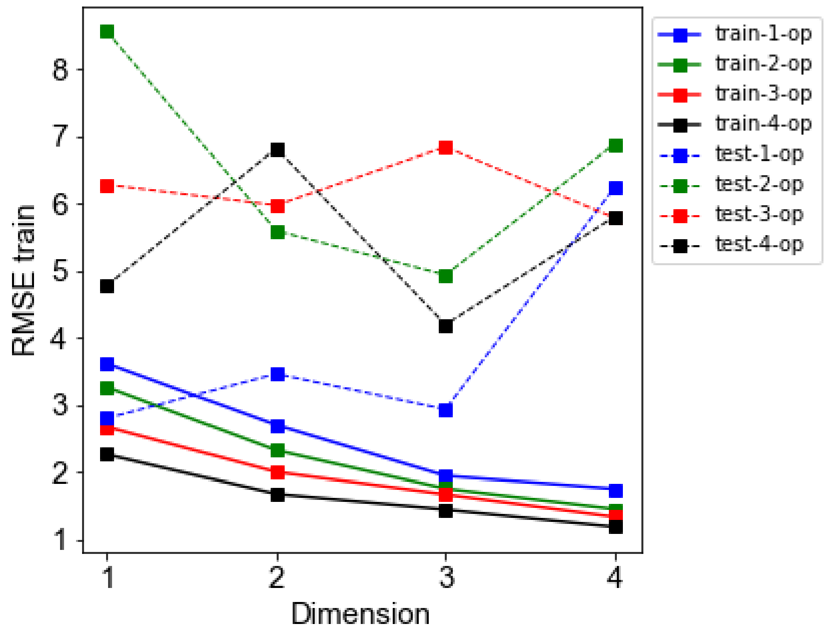
In [ ]:
```
# here only the commented-out code is included
# -> the results for the code will be provided
# -> a scatter plot to show the final results is generated

'''
# empty dictionary for saving CV results
CV_list = {'train_RMSE':[], 'test_RMSE':[], 'R2_train':[],
             'dim':[],'#op':[],'n_subset':[],'feat_per_it':[],
             '#features':[],'test_dopant':[]}

# define dopants for CV
dopants = list(df_1['dopants'].unique())

# model parameters
ops = [1,2,3,4]
dims = [1,2,3,4]
feat_per_it = 40
subset = 170

# select primary features
selected_primary_features = primary_features_Nmax

for dim in dims:
    for op in ops:
        # split train and test to start CV
        for dopant in dopants:
            df_train = df_1[df_1['dopant']!=dopant]
            df_test = df_1[df_1['dopant']==dopant]
            # prepare data
            model_features = selected_primary_features.copy()
            model_features.append(target_property)
            prepared_train = df_train[model_features]
            prepared_test = df_test[model_features]
            # create unit list for all primary features
            # the unit list holds the unit for each of the features in prepared_dat
            unit_list = ['C']
            for i in range(prepared_train.shape[1]-1):
                unit_list.append('eV')
            # create SISSO model for training set
            sisso_model, rmse, features_list  = ST.SISSO(prepared_train, target_pro
                                                ['+','-','^2','^0.5','exp','/',
                           n_operations=op, dim=dim, features_per_iter=feat_per_it
                           n_best_subs=subset)
            # predict values for training set
            train_predict = list(ST.evaluate_sisso(sisso_model, prepared_train, fea
            train_measure = list(prepared_train[target_property])
            # predict values for test set
            test_predict = list(ST.evaluate_sisso(sisso_model, prepared_test, featu
            test_measure = list(prepared_test[target_property])
            # get RMSE test
            error = [pred - meas for (pred,meas) in zip(test_predict, test_measure)
            error_2 = [er**2 for er in error]
            RMSE_test = (sum(error_2)/len(error_2))**0.5
            # save results to dictionary
            CV_list['train_RMSE'].append(round(rmse,2))
            CV_list['test_RMSE'].append(round(RMSE_test,2))
            CV_list['R2_train'].append(r2_score(train_measure,train_predict))
            CV_list['dim'].append(dim)
            CV_list['#op'].append(op)
            CV_list['n_subset'].append(subset)
            CV_list['feat_per_it'].append(feat_per_it)
            CV_list['#features'].append(len(features_list))
            CV_list['test_dopant'].append(dopant)
# save results as dataframe
CV_result_df = pd.DataFrame(CV_list)
CV_result_df.to_excel("CV_results.xlsx")
'''
```

In [15]:
```
CV_result_df = pd.read_excel('CV_results.xlsx')

# plot results
hfont = {'fontname':'arial'}
import matplotlib.font_manager as font_manager
font = font_manager.FontProperties(family='arial',
                                   # weight='bold',
                                   style='normal', size=15)
# save values
train_RMSE = {'1-op':[],'2-op':[],'3-op':[],'4-op':[]}
test_RMSE = {'1-op':[],'2-op':[],'3-op':[],'4-op':[]}

for op in range(len(train_RMSE)):
    temp = CV_result_df[CV_result_df['#op']==op+1]
    for dim in range(len(train_RMSE)):
        temp_2 = temp[temp['dim']==dim+1]

        temp_R2 = st.mean(temp_2['train_RMSE'])
        train_RMSE[list(train_RMSE.keys())[op]].append(temp_R2)
        temp_R2_test = st.mean(temp_2['test_RMSE'])
        test_RMSE[list(test_RMSE.keys())[op]].append(temp_R2_test)
```

```
# plot results summary for training set
colors = ['blue','green','red','black']
dims = [1,2,3,4]
# train
fig, ax = plt.subplots(1,1,figsize=(5,5))
for i,op in enumerate(train_RMSE):
    plt.plot(dims, train_RMSE[op], color=colors[i], marker='s', label=f"train-{op}
ax.legend(prop=font)
plt.xticks([1,2,3,4],[1,2,3,4],size=15,**hfont)
plt.yticks(size=15,**hfont)
plt.xlabel('Dimension',size=15,**hfont)
plt.ylabel('RMSE train',size=15,**hfont)

# plot results summary for testing set
# fig, ax = plt.subplots(1,1,figsize=(5,5))
for i,op in enumerate(test_RMSE):
    plt.plot(dims, test_RMSE[op], color=colors[i], marker='s', label=f"test-{op}",
plt.legend(bbox_to_anchor=(1,1))
```

Out[15]:

```
<matplotlib.legend.Legend at 0x1ffc42a4fd0>
```



4. Generation_of_SISSO_model_with_optimum_

Once the preprocessing method (normalizing values between 0 and 1), and optimal model complexity (3D-1O) are determined, we create respective SISSO model for the study.

In [16]:

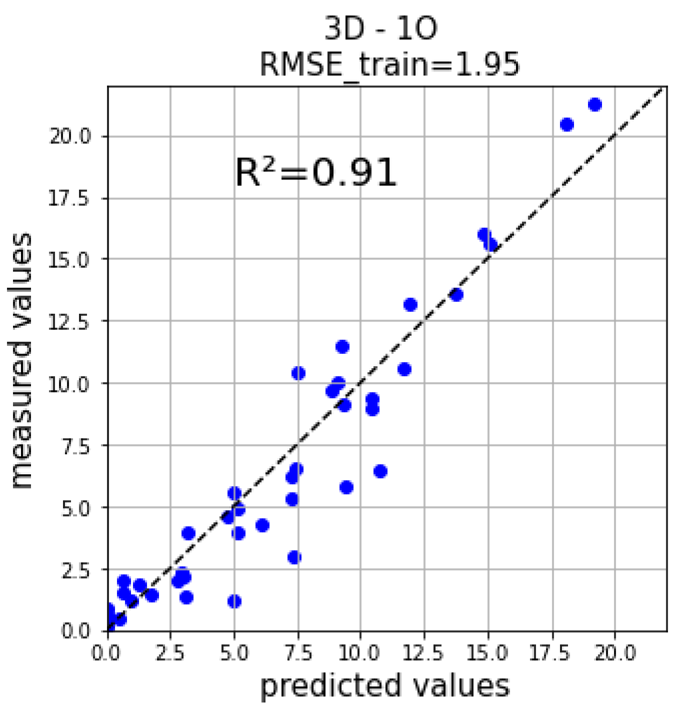
```
# specify optimum preprocessing method
selected_primary_features = primary_features_Nmax
model_type = 'Nmax'

# specify optimum SISSO model complexity
dim = 3
op = 1
feat_per_it = 50
subset = 170
```

```
# create note file to write result
result = open('optimum_SISSO_result_{dim}D-{op}O_.txt','w')

# prepare data
df_model = df_1[df_1['temp'].isin([225,250,275])]
model_features = selected_primary_features.copy()
model_features.append(target_property)
prepared_df = df_model[model_features]

# create unit list for all primary features
# the unit list holds the unit for each of the features in prepared_data
unit_list = ['C']
for i in range(prepared_df.shape[1]-1):
    unit_list.append('eV')

# create SISSO model for training set
sisso_model, rmse, features_list  = ST.SISSO(prepared_df, target_property, unit_li
                                        ['+','-','^2','^0.5','exp','/','*'],
               n_operations=op, dim=dim, features_per_iter=feat_per_it,
               n_best_subs=subset)

# write SISSO result to note
result.write(f"{dim}D, {op}O, {subset}subsets, {feat_per_it}-feats_per_it\n")
result.write(sisso_model.return_models(features_list))
result.write('\n \n')

# predict values for training set
train_predict = list(ST.evaluate_sisso(sisso_model, prepared_df, features_list))
train_measure = list(prepared_df[target_property])

# save prediction results into a separate dataset
r_df = pd.DataFrame({'measured':train_measure, 'predict':train_predict, 'temp':lis
r_df.to_excel('predicted_results.xlsx')

# replace negative prediction with zero
for i in range(len(r_df)):
    if r_df['predict'][i]<0:
        r_df['predict'][i]=0

# create parity scatter plot
fig, ax = plt.subplots(1,1,figsize=(5,5))
plt.scatter(r_df['predict'], r_df['measured'], color='blue')

plt.xlabel('predicted values', size=15)
plt.ylabel('measured values', size=15)
plt.xlim(0,22)
plt.ylim(0,22)
plt.grid()
plt.axline((0, 0), slope=1, color="black", linestyle='--')
plt.title(f'{dim}D - {op}O \n RMSE_train={round(rmse,2)}', size=15)
plt.text(x=5,y=18,s=f"R²={round(r2_score(r_df['measured'], r_df['predict']),2)}",
```

```
          RMSE            Model
1D:     3.634972        + 51.624 ((temp_Nmax)/(CO_Cl[C+][O-]_29.0_250_Sh_Nmax)) -
38.424
2D:     2.812140        + 48.568 ((temp_Nmax)/(CO_Cl[C+][O-]_29.0_250_Sh_Nmax)) -
24.446 ((CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac1_Sh_Nmax)/(CO_S1SC(=O)1_29.0_Ovac0_Sh_Nm
ax)) - 11.389
3D:     1.945028        + 53.980 ((temp_Nmax)*(CO2_ClC([O-])=O_25.0_290_Sh_Nmax))
+ 7.057 ((CO2_ClO[C+](=O)_29.0_250_Sh_Nmax)/(CO+OH_Cl[C+][O-]_25.0_250_Sh_Nmax)) -
29.041 ((CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac1_Sh_Nmax)/(CO_S1SC(=O)1_29.0_Ovac0_Sh_Nm
ax)) - 18.399
```

```
C:\Users\m_khatamirad\AppData\Local\Temp\ipykernel_11456\547249723.py:48: SettingW
ithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stabl
e/user_guide/indexing.html#returning-a-view-versus-a-copy
  r_df['predict'][i]=0
```

Out[16]:
```
Text(5, 18, 'R²=0.91')
```



5. Subgroup_discovery

In [ ]:
```
The generated descriptors in the SISSO model are used to detect outstanding perfor
```

In [25]:
```python
import os
os.chdir('C:\\_Mohammad\\__Publications\\_3-volcano_CO2-MeOH\\_processed_data\\202
# primary features which were used in SISSO model
A = 'CO2_ClC([O-])=O_25.0_290_Sh_Nmax'
B = 'CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac1_Sh_Nmax'
C = 'CO_S1SC(=O)1_29.0_Ovac0_Sh_Nmax'
D = 'CO2_ClO[C+](=O)_29.0_250_Sh_Nmax'
E = 'CO+OH_Cl[C+][O-]_25.0_250_Sh_Nmax'

# replace descriptors with letters
df_1['A'],df_1['B'],df_1['C'],df_1['D'],df_1['E']=df_1[A],df_1[B],df_1[C],df_1[D],

# secondary descriptors from SISSO model
df_1['F'] = df_1[B]/df_1[C]
df_1['G'] = df_1[D]/df_1[E]
df_1['H'] = df_1[A]*df_1['temp_Nmax']

# primary features used for SGD
# these features are descriptors used in SISSO model
primary_features = ['F','G','A','B','C','D','E','temp_Nmax']
target_feature = ['Y_MeOH']

#create a dataframe with the input data to realkd
df_input = df_1[primary_features+target_feature].copy()
```

```
# write the input file for realkd
id_job='MeOH'
n_clusters=20
n_seeds=10000
n_results=1000

#threshold defining what is considered near-optimal
threshold=0.60

#check if old files exist for the specified job_id and remove them
dirpath = os.path.join('output', id_job)
if os.path.exists(dirpath) and os.path.isdir(dirpath):
    shutil.rmtree(dirpath)

SGD.write_input('./',
            df_input,
            id_job,
            n_clusters,
            'EMM_SAMPLER',
            'cumulative_jensen_shannon_divergence',
            n_results,
            n_seeds,
            target_feature[0])

file_results='MeOH_results.json'

os.system('java -jar realkd-0.7.2-jar-with-dependencies.jar '+id_job+'.json')
file_results='./output/'+id_job+'/'+os.listdir('./output/'+id_job+'/')[0]+'/result

# iteration parameters
# ==============================================================================

kf=KFold(n_splits=10, random_state=10, shuffle=True)

df_input_reset = df_input.reset_index()
df_it = df_input_reset.drop(columns=['index'])

iteration_result = {'primary_features':primary_features, 'target_feature':target_f

SGD_result = {'iteration':[],'Q':[], f"{target_feature[0]}":[]}
for D in primary_features:
    SGD_result[D] = []

indices_test=[]
for train_index,test_index in kf.split(df_it):
    indices_test.append(test_index)

# number of iterations
iteration_number = 10

# start iteration
# ==============================================================================

for it in range(iteration_number):
    df_train=df_it.drop(indices_test[it])
    id_job='MeOH'+'_cv_'+str(it)

    #check if old files exist for the specified job_id and remove them
    dirpath = os.path.join('output', id_job)
    if os.path.exists(dirpath) and os.path.isdir(dirpath):
        shutil.rmtree(dirpath)

    SGD.write_input('./',
                df_train,
                 id_job,
                 n_clusters,
                 'EMM_SAMPLER',
                 'cumulative_jensen_shannon_divergence',
                 n_results,
                 n_seeds,
                 target_feature[0])

    #here we load the output files of this analysis
    file_results='MeOH_cv_iteration_'+str(it)+'.json'

    #uncomment the two lines below for running the SGD analysis
    os.system('java -jar realkd-0.7.2-jar-with-dependencies.jar '+id_job+'.json')
    file_results='./output/'+id_job+'/'+os.listdir('./output/'+id_job+'/')[0]+'/re


    with open(file_results) as json_file:
        data = json.load(json_file)
        sg_max_Q=SGD.get_sg_info(data,0)
        Q_max=sg_max_Q[0]*sg_max_Q[1]
        coverage_list=[SGD.get_sg_info(data,i)[0] for i in range(len(data))]
        utility_function_list=[SGD.get_sg_info(data,i)[1] for i in range(len(data)
        coverage_list_selected=[]
        utility_function_list_selected=[]
        for i in range(len(data)):
            sg_info=SGD.get_sg_info(data,i)
            if sg_info[0]*sg_info[1] >= threshold*Q_max:
                coverage_list_selected.append(sg_info[0])
                utility_function_list_selected.append(sg_info[1])
        max_utility=max(utility_function_list_selected)
        list_index_max_utility_function=[i for i, j in enumerate(utility_function_
        sg_max_u=SGD.get_sg_info(data,list_index_max_utility_function[0])

    df_test=pd.concat([df_it,df_train]).drop_duplicates(keep=False)
    df_test_selected=df_test.query(sg_max_u[3])


    iteration_result[f'iteration_{it+1}'] = [
        {'rule':sg_max_u[3]} , {f'\n average {target_feature[0]} in the test set '
        {f'\n average {target_feature[0]} in the test set SG ':"{:.3f}".format(np.
        ]

# save iteration results
# ============================================================================

    # get rules, detected features, and their indices
    rules , Ds = sg_max_u[3], sg_max_u[4]
    indices = [rules.index(d) for d in Ds]

    # add iteration number, quality function value, avg target feature value
    SGD_result['iteration'].append(it+1)
    SGD_result['Q'].append(sg_max_u[0]*sg_max_u[1])
    SGD_result[f"{target_feature[0]}"].append(sg_max_u[2])

    # add detected features and their rules
    for i in range(len(indices)):
        try:
            SGD_result[Ds[i]].append(rules[indices[i]:indices[i+1]-3])
        except:
            SGD_result[Ds[i]].append(rules[indices[-1]:])

    # add empty val for undetected features
    undetected = [item for item in primary_features if item not in Ds]
    for d in undetected:
        SGD_result[d].append('-')


# write iteration results to a file and save
with open('iteration_results.txt','w') as result_file:
    for item in list(iteration_result.keys()):
        result_file.write(f"{item}: {iteration_result[item]} \n")

# save SGD iteration results to dataframe
# ============================================================================
# remove empty entries in result dictionary
# emoty entres are generated if a rule has more than one condition
for rule in SGD_result:
    if len(SGD_result[rule])>10:
        temp = SGD_result[rule]
        temp_2 = [val for val in temp if val!='']
        SGD_result[rule]=temp_2

# Sheet-1: save SGD iteration result in dataframe
SGD_result_df = pd.DataFrame(SGD_result)
SGD_result_df.to_excel('SGD_output.xlsx')

SGD_result_df
```

Out[25]:

	iteration	Q	Y_MeOH	F	G	A	B	C	D	
0	1	0.052379	18.853333	-	-	-	-	-	D>=0.881	E<0.
1	2	0.050366	17.988000	F<=1.1169	G>=1.2128	A>=0.8903	-	C<0.8836	-	
2	3	0.044506	18.896000	-	G>1.693	A>=0.9578	-	-	-	
3	4	0.052107	18.853333	-	G>=1.297	A>=0.8903	-	-	-	E<0.
4	5	0.047930	19.218000	F<0.9744	-	A>=0.9214	-	-	-	E<0.
5	6	0.037225	18.967500	-	G>1.693	A>=0.9115	-	-	-	
6	7	0.046503	20.375000	F<=1.0162	G>1.693	-	-	-	-	
7	8	0.051908	18.853333	-	-	A>=0.9578	-	-	-	E<0.
8	9	0.045872	20.375000	-	-	A>=0.9623	-	C<0.8836	-	
9	10	0.038146	21.030000	-	G>1.693	A>=0.9427	-	-	-	

6. Overlay_of_SISSO_prediction_measured_value

Generated boundaries of model descriptors, are overlayed with distribution of their values for the catalysts used in the study. This evaluates how distant each of the tested catalysts are from the detected limits for outstanding performance.

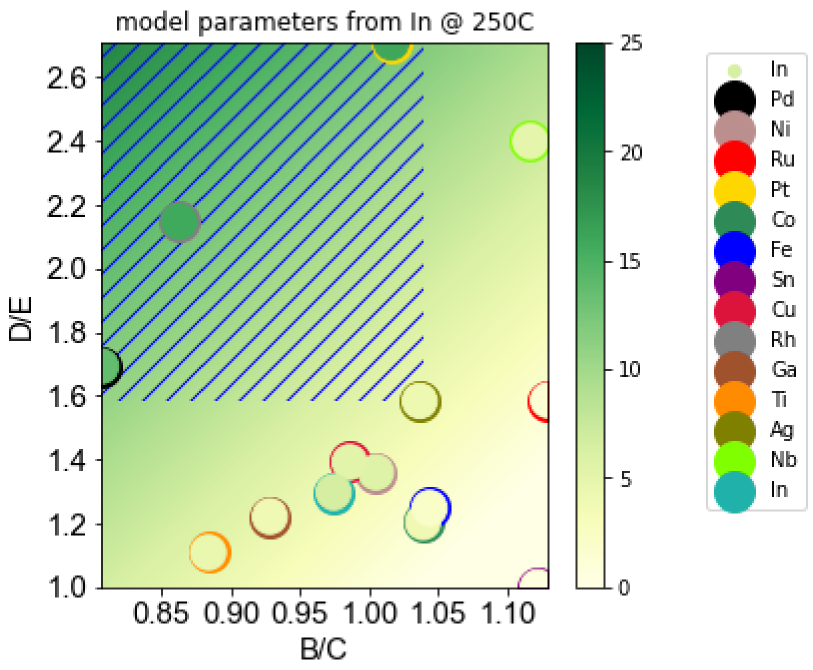
Additionally, by plotting the measured value and the SISSO-predicted value next to each other, we evaluate the accuracy of the SISSO model once more.

In [23]:

```
# define SISSO descriptors
A = 'CO2_ClC([O-])=O_25.0_290_Sh_Nmax'
B = 'CHO2_S1SC(=O)[O+]1[H+]_29.0_Ovac1_Sh_Nmax'
```

```
C = 'CO_S1SC(=O)1_29.0_Ovac0_Sh_Nmax'
D = 'CO2_ClO[C+](=O)_29.0_250_Sh_Nmax'
E = 'CO+OH_Cl[C+][O-]_25.0_250_Sh_Nmax'
F = 'F'
G = 'G'

# specify colors for assigning to catalysts
colors = ['black','rosybrown','red','gold','seagreen','blue','purple','crimson','g
          'olive','chartreuse','lightseagreen','steelblue','navy']

# generate scatter plot parameters
DOPANTS = df_1['dopant'].unique()

# specify catalyst for which you want to read constant values
dopants = ['In']

# generate plot
fig, ax = plt.subplots(1,1,figsize=(5,5))

# select promoter for model
for promoter in dopants:

    # select temperature (0: 225, 1:250, 2:275)
    Ts = [225, 250, 275]
    j=1
    T_model = list(df_1['temp_Nmax'].unique())[j]

    # the values which should be set to a constant
    SISSO_params = [A]
    SISSO_params_vals = {}
    for param in SISSO_params:
        temp = df_1[df_1['dopant']==promoter]
        SISSO_params_vals[param] = (list(temp[param].unique()))[0]

    # insert SISSO equation
    x = np.arange(min(df_1['F']), max(df_1['F']),0.001)
    y = np.arange(min(df_1['G']), max(df_1['G']),0.001)
    x,y = np.meshgrid(x, y)

    Z = 49.482*T_model*SISSO_params_vals[A] + 7.057*(y) - 29.041*(x) - 18.399

    # 2D plot
    # 1) plot SISSO prediction
    plt.scatter(x,y, c=Z, vmin=0, vmax=25, cmap='YlGn', label=promoter)
    plt.xlabel('B/C', size=15,**hfont)
    plt.ylabel('D/E', size=15,**hfont)
    plt.xticks(size=15,**hfont)
    plt.yticks(size=15,**hfont)
    # fill the constraints from SGD result
    plt.fill_between([0, 1.0392], [3, 3],[1.5839,1.5839], hatch='//',color='none',
    # set axis lim
    plt.xlim(min(df_1['F']), max(df_1['F']))
    plt.ylim(min(df_1['G']), max(df_1['G']))
    # 2) plot B/C and D/E for catalysts
    for i,dop in enumerate(DOPANTS):
        temp = df_1[(df_1['dopant']==dop)&(df_1['temp_Nmax']==T_model)]
        plt.scatter(temp['F'], temp['G'], label=dop, s=400, color=colors[i])
    # plt.legend(bbox_to_anchor=(2, 1.02))
    plt.title(f"model parameters from {promoter} @ {Ts[j]}C")
    # 3) plot measured value for each scatter point
    TEMP = df_1[df_1['temp_Nmax']==T_model]
    plt.scatter(TEMP['F'], TEMP['G'], c=TEMP['Y_MeOH'], s=300, zorder=1, vmin=0, v
plt.colorbar()
plt.legend(bbox_to_anchor=(1.6,1))
```

Out[23]:
```
<matplotlib.legend.Legend at 0x1ffc2c46880>
```



In [ ]: