Machine Learning for Core Engineering Disciplines

Instructor:
Prof. Ananth Govind Rajan
Department of Chemical Engineering
Indian Institute of Science, Bengaluru
Email: ananthgr@iisc.ac.in
Website: https://agrgroup.org

2. Linear regression using Python

2.1 Simple linear regression

One often stores tabular data in comma separated value (CSV) files. These files can be opened by Microsoft Excel. Within Python, one can use the Pandas package to read CSV (.csv) and Excel (.xls/.xlsx) files.

See, e.g., the below commands to read an Excel file named "Physical_Feature_Data.xlsx" containing data regarding the physical features for 20840 nanopore shapes in graphene, a two-dimensional (2D) material, from the following research article:

Sheshanarayana, R.; Govind Rajan, A. Tailoring Nanoporous Graphene via Machine Learning: Predicting Probabilities and Formation Times of Arbitrary Nanopore Shapes. *J. Chem. Phys.* **2022**, *156*, 204703.

The Excel file was made available in the previous tutorial.

```
In [1]: import pandas as pd
```

```
In [5]: pd.read_excel('/Users/ananthgr/Documents/Acads/IISc/Courses/CH 251/Physical_Feature_Data.xlsx')
```

Out[5]:

	num_removed_atoms	num_ZZ_atoms	num_AC_atoms	num_UA_atoms	num_SB_atoms	count_rim_atoms	num_incomplete_hexagons	num_5_membered
0	4	6	0	0	0	18	6	
1	4	4	2	0	0	18	6	
2	4	6	0	0	0	18	6	
3	5	5	2	0	0	20	7	
4	5	5	0	0	1	19	7	
...	...	...	...	...	...	...	...	
20835	22	8	8	2	0	42	21	
20836	22	7	6	1	1	37	20	
20837	22	6	8	0	1	37	20	
20838	22	8	8	0	0	38	20	
20839	22	10	4	0	1	37	20	

20840 rows × 24 columns

The data in the Excel file can also be stored in a data structure, as follows:

```
In [6]: data=pd.read_excel('/Users/ananthgr/Documents/Acads/IISc/Courses/CH 251/Physical_Feature_Data.xlsx')
```

The type of the data structure can be seen as follows:

```
In [7]: type(data)
Out[7]: pandas.core.frame.DataFrame
```

Finally, one can determine the number of rows (data points) and columns (features) in the dataset as follows:

```
In [8]: data.shape
Out[8]: (20840, 24)
```

Note that pressing tab after writing the name of a variable/package shows a list of commands that can be executed.

One can see that there are 20,840 data points in the dataset. Furthermore, there are 24 columns in the data set.

The first 22 columns contain the features of the graphene nanopores and the last 2 columns contain, respectively, the formation time and probability of formation of each nanopore. Thus, there are 2 target variables and 22 features in our dataset. Some of these features are depicted pictorially in Figure 1 below.

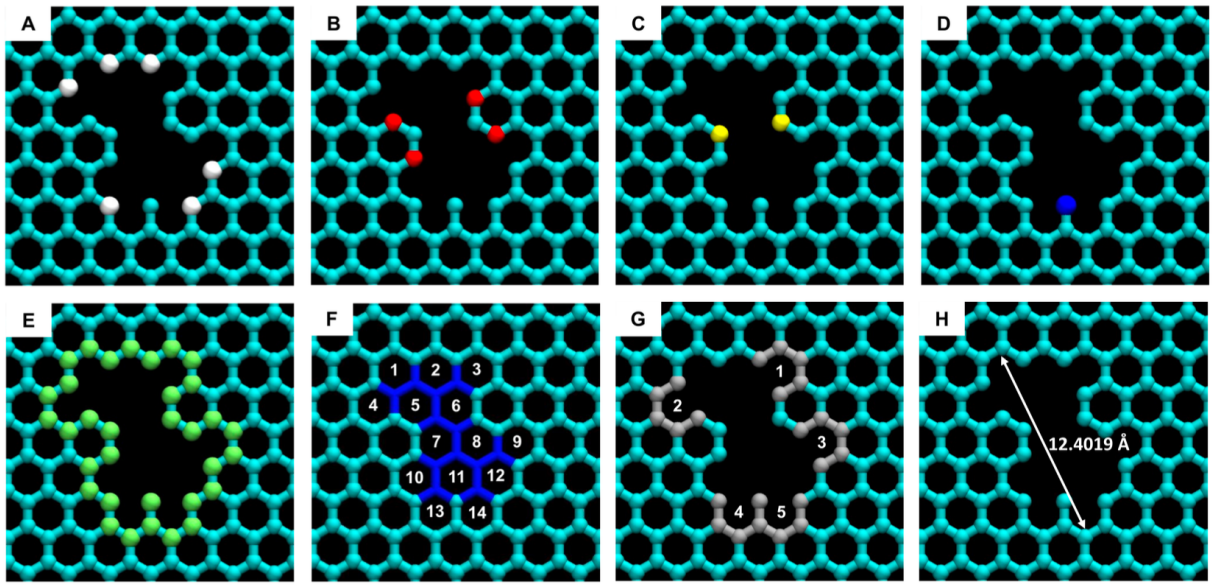


Figure 1. Out of the 22 features considered above, 8 have been depicted above with the help of one of the $N = 12$ nanopores, i.e., a nanopore formed by removing 12 atoms from the graphene lattice. A. Number of zigzag atoms, shown in white ($n_{zz} = 6$), B. Number of armchair atoms, shown in red ($n_{ac} = 2$), C. Number of corner atoms, shown in yellow ($n_{cor} = 2$), D. Number of singly bonded atoms, shown in blue ($n_{sb} = 1$), E. Number of rim atoms, shown in green ($n_{rim} = 33$), F. Number of incomplete hexagons in blue, numbered from 1 to 14 ($n_{hex} = 14$), G. Number of five-membered

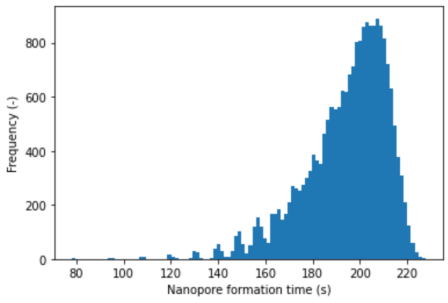
rings shown in silver, numbered from 1 to 5 ($n_{5m} = 5$), and, H. the maximum possible distance between any two rim atoms ($d_{max} = 12.4019$ A).

Plotting data in Python is made simple by the use of the Matplotlib package. For this purpose, it is useful to load the Pyplot module from Matplotlib.

```
In [44]: import matplotlib.pyplot as plt
```

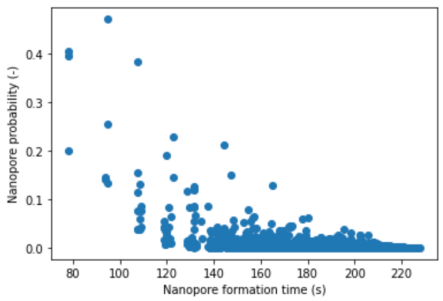
Subsequently, several types of plots can be made. Our first example is that of a bar plot:

```
In [49]: plt.hist(data['time'],bins=100)
         plt.xlabel('Nanopore formation time (s)')
         plt.ylabel('Frequency (-)')
Out[49]: Text(0, 0.5, 'Frequency (-)')
```



Another example is that of a scatter plot:

```
In [53]: plt.scatter(data['time'],data['probability'])
         plt.xlabel('Nanopore formation time (s)')
         plt.ylabel('Nanopore probability (-)')
Out[53]: Text(0, 0.5, 'Nanopore probability (-)')
```



We now proceed to train a simple linear regression model using the dataset provided to predict the formation time and the probabilities of nanopore shapes based on the available features. To this end, we first import the numpy package as well as the linear regression module from the Scikit-learn package, as follows:

```
In [9]: import numpy as np
        from sklearn.linear_model import LinearRegression
```

Next, we drop the dependent variables from the dataset and obtain the matrix of the datapoints, X, as follows:

```
In [11]: X=data.drop(columns=['time','probability'])
```

We also obtain the vectors of the two target variables using the following command:

```
In [40]: y1=data['time']
         y2=data['probability']
```

Subsequently, a simple linear regression model can be trained for the formation time, and its R^2 value can be checked, as follows:

```
In [8]: reg1 = LinearRegression().fit(X, y1)

In [10]: reg1.score(X,y1)
Out[10]: 0.9312836016269951
```

It is also possible to obtain the coefficients and the intercept for the linear regression model:

```
In [43]: reg1.coef_
Out[43]: array([ 7.88944044e+00,  6.14014236e+00,  5.81609852e+00,  5.01547714e+
         00,
                1.03772563e+01, -6.59954124e-01, -7.06673212e+00,  1.31497606e+
         00,
                1.55757327e+00,  3.77240688e+01,  1.95399252e-14, -3.73170719e+
         00,
               -1.97323035e+00, -7.63199594e-01, -1.97323035e+00, -3.73170719e+
         00,
               -7.83410014e-01, -3.03800115e+00,  1.81699722e-01, -1.13141970e+
         00,
               -1.19713520e+00, -1.11405708e+00])

In [44]: reg1.intercept_
Out[44]: 54.378063358970024
```

Now, based on your knowledge of Python and what was taught in class, can you write your own code to predict the confidence intervals for the coefficients in the linear regression model? You may find the scipy.special.stdtrit (student's t inverse transform) function useful.

One can also determine other performance metrics for the model, such as the mean absolute error (MAE) and the root-mean-square error (RMSE) as follows:

```
In [17]: ypred1=reg1.predict(X)

In [19]: import sklearn
         sklearn.metrics.mean_absolute_error(y1,ypred1)
Out[19]: 3.6823636672504665

In [20]: np.sqrt(sklearn.metrics.mean_squared_error(y1,ypred1))
Out[20]: 4.570917600429209

In [50]: sklearn.metrics.r2_score(y1,ypred1)
Out[50]: 0.9312836016269951
```

Recall that, in the class, we discussed the splitting of the dataset into training and testing sets. How is that accomplished? This can be done using the following command:

```
In [61]: X1_train, X1_test, y1_train, y1_test=sklearn.model_selection.train_test_split(X,y1,test_size=0.25,random_state=42)
```

where random_state is simply a seed to determine the random splitting of the dataset. The above commands randomly selects 25% datapoints to be in the test set and remaining 75% datapoints to be in the training set. We can now train the model using the training set and check the performance metrics using the test set, as follows:

```
In [23]: reg1_train=LinearRegression().fit(X1_train, y1_train)

In [24]: reg1_train.coef_
Out[24]: array([ 8.01396333e+00,  6.67718701e+00,  6.37138895e+00,  5.47665375e+
         00,
                1.12877823e+01, -8.34760546e-01, -7.35829550e+00,  1.27230859e+
         00,
                1.57496361e+00,  3.87849555e+01,  3.55271368e-15, -3.23081035e+
         00,
               -1.73779622e+00, -4.31277571e-01, -1.73779622e+00, -3.23081035e+
         00,
               -8.41425828e-01, -4.50826613e+00,  5.97841442e-01, -1.76590678e+
         00,
               -1.67060053e+00, -7.30781163e-01])

In [25]: y1_test_pred=reg1_train.predict(X1_test)
         y1_train_pred=reg1_train.predict(X1_train)

In [26]: sklearn.metrics.r2_score(y1_test,y1_test_pred)
Out[26]: 0.9302565469293936

In [27]: sklearn.metrics.mean_absolute_error(y1_test,y1_test_pred)
Out[27]: 3.6491895802025485
```

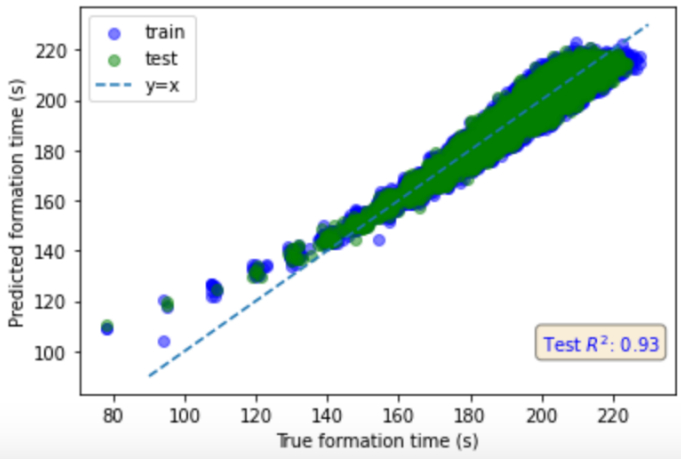
Note that the R^2 metric on the test set is 0.93. However, in the original paper, the R^2 value for a complex two-stage model is 0.95. So, why did the authors choose a more complex model, when a linear regression model is providing an "acceptable" R^2 metric? Let us check the parity plot for the train and the test data:

```
In [28]: import matplotlib.pyplot as plt
```

```
In [29]: plt.scatter(y1_train,y1_train_pred,c="blue",alpha=0.5)
         plt.scatter(y1_test,y1_test_pred,c="green",alpha=0.5)
         plt.xlabel('True formation time (s)')
         plt.ylabel('Predicted formation time (s)')
         parity_x = np.linspace(90,230,100)
         plt.plot(parity_x,parity_x,'--')
         plt.legend(["train","test","y=x"])

         train_mae = sklearn.metrics.mean_absolute_error(y1_train,y1_train_pred)
         train_r2=sklearn.metrics.r2_score(y1_train,y1_train_pred)
         test_mae=sklearn.metrics.mean_absolute_error(y1_test,y1_test_pred)
         test_r2=sklearn.metrics.r2_score(y1_test,y1_test_pred)

         print('Train MAE = ', train_mae)
         print('Train R^2 = ', train_r2)
         print('Test MAE  = ', test_mae)
         print('Test R^2  = ', test_r2)

         box_style=dict(boxstyle='round', facecolor='wheat', alpha=0.5)
         plt.text(200,100,"Test $R^2$: {:.2f}".format(round(test_r2,2)),{'color':
```

```
Train MAE =  3.695422088710631
Train R^2 =  0.9315622129258776
Test MAE  =  3.6491895802025485
Test R^2  =  0.9302565469293936
```

```
Out[29]: Text(200, 100, 'Test $R^2$: 0.93')
```



One can compare the obtained parity plot with the corresponding one reported in the original work, reproduced as Figure 2b below.

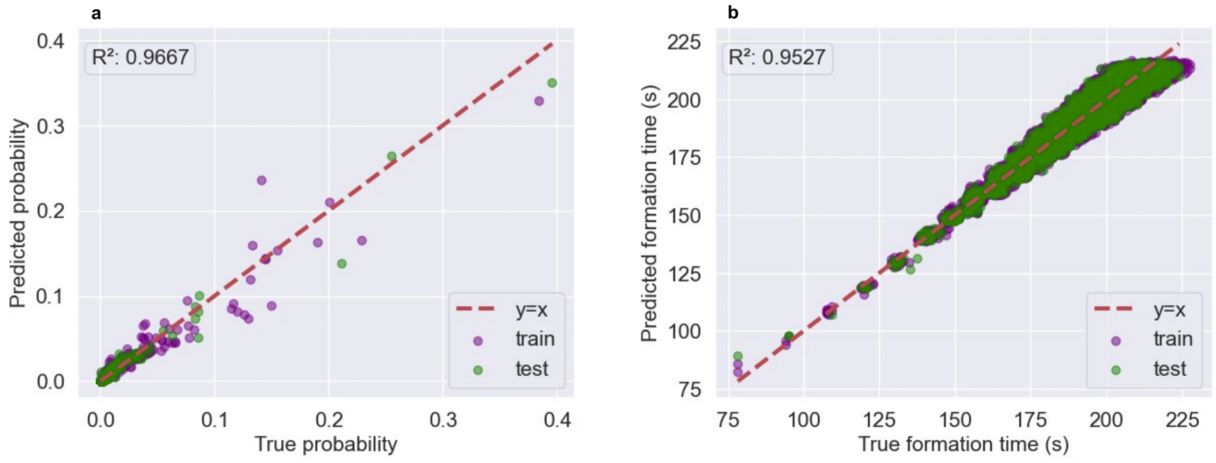
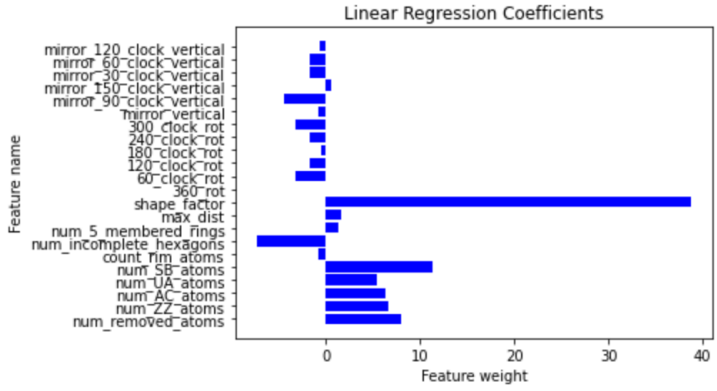


Figure 2. Parity plots for the target variables. A represents the parity plot for the true and predicted probabilities in the training (purple) and test (maroon) sets, while B represents the same for the formation times. The panel at the top-left corner of both the plots represents the R^2 scores, using the test set, achieved by the two-stage models to predict the respective targets.

Immediately, one sees that the simple linear regression model is not at all able to accurately predict the formation times of the nanopores with low formation times, thus justifying the more complex models used in the original work.

Now, what if one wanted to pictorially represent the relative values of the coefficients in the linear regression model? This can be achieved using the following piece of code:

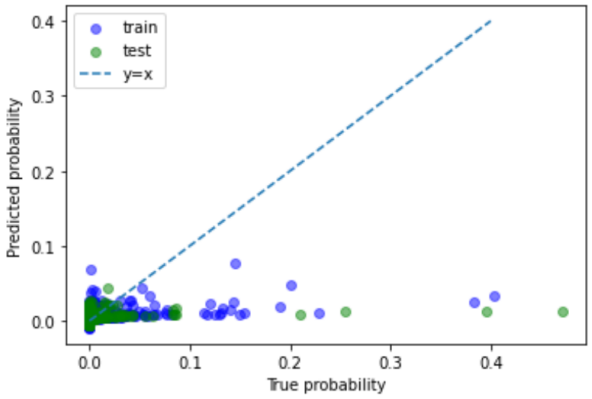
```
In [133]: array = np.arange(1, 23)
          # Plot the coefficients
          plt.barh(array,reg1_train.coef_, color='blue')
          plt.xlabel('Feature weight')
          plt.ylabel('Feature name')
          plt.title('Linear Regression Coefficients')
          plt.yticks(array,list(X))

          # Show the plot
          plt.show()
```



It is certainly possible to change the labels given on the vertical axis, and one should do that to make the plot more presentable.

Can you now train a simple linear regression model for the prediction of the nanopore probabilities? The results may surprise you! (You need to write your own code for this part.)

```
Train MAE =  0.0017127152591797745
Train R^2 =  0.14784626281452806
Test MAE  =  0.001822033006627761
Test R^2  =  0.050812635169134146
```



2.2 Ridge regression

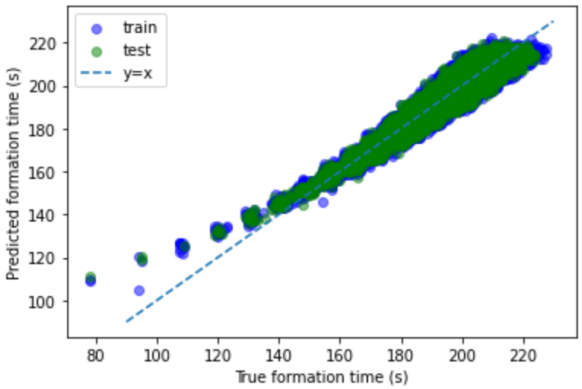
Perhaps, one may think, can we improve the performance of the model by trying ridge regression? This can be implemented as follows:

```
In [30]: from sklearn.linear_model import Ridge

In [31]: reg1_ridge = Ridge(alpha=1.0)
         reg1_ridge.fit(X1_train,y1_train)
         Ridge()

Out[31]: ▾ Ridge
         Ridge()
```

However, one can see that the performance of the model is not much better:

```
[n [32]: y1_ridge_train_pred=reg1_ridge.predict(X1_train)
         y1_ridge_test_pred =reg1_ridge.predict(X1_test)
         plt.scatter(y1_train,y1_ridge_train_pred,c="blue",alpha=0.5)
         plt.scatter(y1_test,y1_ridge_test_pred,c="green",alpha=0.5)
         plt.xlabel('True formation time (s)')
         plt.ylabel('Predicted formation time (s)')
         parity_x = np.linspace(90,230,100)
         plt.plot(parity_x,parity_x,'--')
         plt.legend(["train","test","y=x"])

         train_mae = sklearn.metrics.mean_absolute_error(y1_train,y1_ridge_train_
         train_r2=sklearn.metrics.r2_score(y1_train,y1_ridge_train_pred)
         test_mae=sklearn.metrics.mean_absolute_error(y1_test,y1_ridge_test_pred)
         test_r2=sklearn.metrics.r2_score(y1_test,y1_ridge_test_pred)

         print('Train MAE = ', train_mae)
         print('Train R^2 = ', train_r2)
         print('Test MAE  = ', test_mae)
         print('Test R^2  = ', test_r2)
```

```
Train MAE =  3.6971793580915295
Train R^2 =  0.9314748351671592
Test MAE  =  3.6485288942387015
Test R^2  =  0.9302377893356745
```



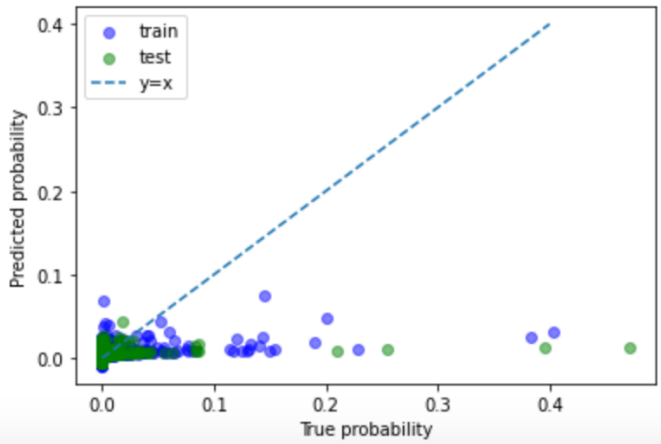
Note that alpha is the weight given to the sum of squares of the coefficients in the loss function. The same observation can be verified for the prediction of the nanopore probabilities too.

One way to improve the model is by developing a two-stage procedure wherein a classification model is first used to find whether the formation time is low or not, and then a separate regression model is used for each class. You can try this on your own.

```
In [109]: reg2_ridge = Ridge(alpha=1.0)
          reg2_ridge.fit(X2_train,y2_train)
          Ridge()

          y2_ridge_train_pred=reg2_ridge.predict(X2_train)
          y2_ridge_test_pred =reg2_ridge.predict(X2_test)
          plt.scatter(y2_train,y2_ridge_train_pred,c="blue",alpha=0.5)
          plt.scatter(y2_test,y2_ridge_test_pred,c="green",alpha=0.5)
          plt.xlabel('True probability')
          plt.ylabel('Predicted probability')
          parity_x = np.linspace(0,0.4,100)
          plt.plot(parity_x,parity_x,'--')
          plt.legend(["train","test","y=x"])

          train_mae = sklearn.metrics.mean_absolute_error(y2_train,y2_ridge_train_
          train_r2=sklearn.metrics.r2_score(y2_train,y2_ridge_train_pred)
          test_mae=sklearn.metrics.mean_absolute_error(y2_test,y2_ridge_test_pred)
          test_r2=sklearn.metrics.r2_score(y2_test,y2_ridge_test_pred)

          print('Train MAE = ', train_mae)
          print('Train R^2 = ', train_r2)
          print('Test MAE  = ', test_mae)
          print('Test R^2  = ', test_r2)
```

The results are given below:

```
Train MAE =  0.0017131432509949734
Train R^2 =  0.14775854971058855
Test MAE  =  0.0018205405451661308
Test R^2  =  0.051696928722665625
```



3. Unraveling correlations in the dataset using Seaborn

We will now learn a few techniques to conveniently visualize the relationships that exist between the various variables in our dataset. To this end, we will make use of the Seaborn package for Python that allows for data visualization using Matplotlib. We first install Seaborn using the familiar pip install command:

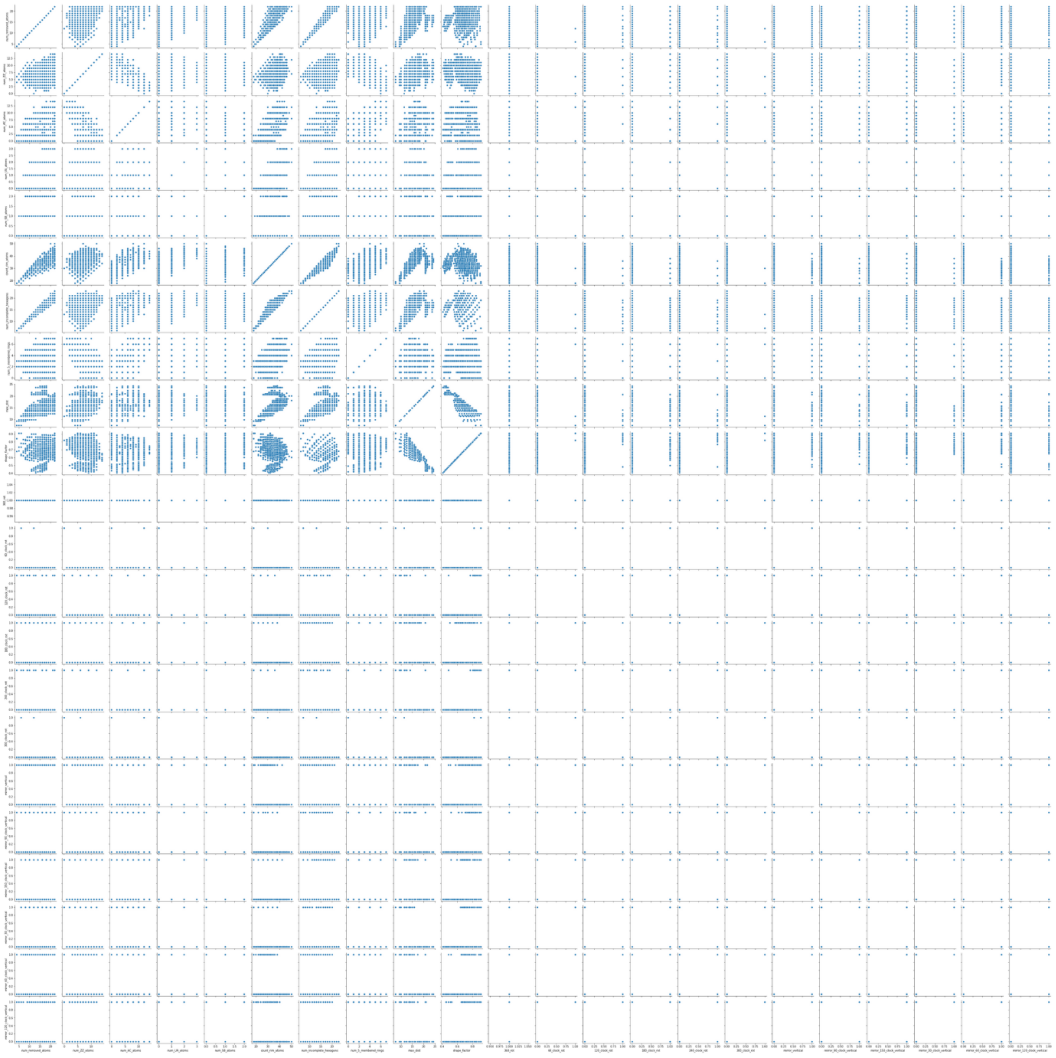
```
In [115]: pip install seaborn
```

One can now create a grid of *y* vs. *x* plots for each variable in the dataset as follows:

```
In [119]: import seaborn as sn
          g = sn.PairGrid(X)
          g.map(sn.scatterplot)
```

Zooming into the plots gives an idea of how each pair of variables are distributed in 2D space.

```
Out[119]: <seaborn.axisgrid.PairGrid at 0x2a409b1c0>
```



It is also possible to determine the correlation coefficients between each pair of variables in the dataset and create a heatmap using Seaborn:

In [181]:
```
data.corr()
```

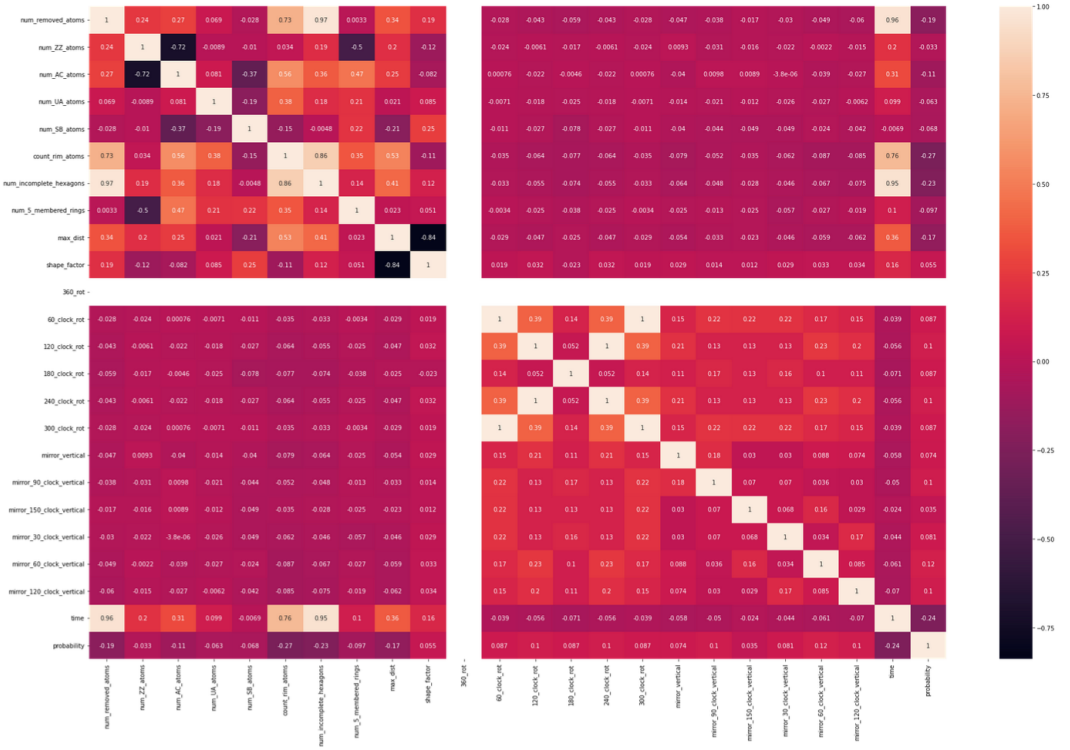
Out[181]:

	num_removed_atoms	num_ZZ_atoms	num_AC_atoms	num_UA_atoms
num_removed_atoms	1.000000	0.238644	0.272422	0.068730
num_ZZ_atoms	0.238644	1.000000	-0.720916	-0.008850
num_AC_atoms	0.272422	-0.720916	1.000000	0.080993
num_UA_atoms	0.068730	-0.008850	0.080993	1.000000
num_SB_atoms	-0.027981	-0.010195	-0.373787	-0.191023
count_rim_atoms	0.725099	0.033973	0.564003	0.383665
num_incomplete_hexagons	0.972075	0.186952	0.362811	0.177523
num_5_membered_rings	0.003291	-0.496400	0.467237	0.211446
max_dist	0.339033	0.200017	0.247555	0.020733
shape_factor	0.190528	-0.120861	-0.082092	0.084856
360_rot	NaN	NaN	NaN	NaN
60_clock_rot	-0.028320	-0.023794	0.000763	-0.007088
120_clock_rot	-0.042972	-0.006116	-0.022213	-0.018076
180_clock_rot	-0.058776	-0.016696	-0.004607	-0.024620
240_clock_rot	-0.042972	-0.006116	-0.022213	-0.018076
300_clock_rot	-0.028320	-0.023794	0.000763	-0.007088
mirror_vertical	-0.047087	0.009293	-0.040487	-0.014439
mirror_90_clock_vertical	-0.037552	-0.030829	0.009846	-0.020712
mirror_150_clock_vertical	-0.017473	-0.015660	0.008858	-0.011647
mirror_30_clock_vertical	-0.029954	-0.021547	-0.000004	-0.025815
mirror_60_clock_vertical	-0.049360	-0.002189	-0.039263	-0.027305
mirror_120_clock_vertical	-0.059601	-0.015127	-0.027365	-0.006241
time	0.957393	0.204877	0.305127	0.099074
probability	-0.185887	-0.032733	-0.108048	-0.062824

In [190]:
```
corr_matrix = data.corr()
plt.figure(figsize = (32,20))
f=sn.heatmap(corr_matrix, annot=True)
```

The resultant heatmap is provided below:

```
In [190]: corr_matrix = data.corr()
          plt.figure(figsize = (32,20))
          f=sn.heatmap(corr_matrix, annot=True)
```



Which variables have the highest correlation coefficients and which ones have the lowest? Are there any unexpected correlations that you are able to deduce from the heat map?

4. Cross validation for linear regression and LASSO

4.1 Introduction to the dataset

In this part of the tutorial, we will explore the use of cross validation to optimize the prediction of the hydrogen adsorption free energy (ΔG_H) on the $Ni_2P(0001)$ surface doped by various nonmetal dopants. The ΔG_H value is a good descriptor for the catalytic activity of any material for the hydrogen evolution reaction (HER) involved in electrochemical water splitting. As you may know, water splitting is a carbon dioxide (CO_2)-free route to produce hydrogen for use in various industries, such as hydrotreating of petroleum, ammonia production, and methanol synthesis. Presently, most of the hydrogen used around the world is produced from steam methane reforming which releases a significant amount of CO_2 into the Earth's atmosphere, thus contributing to global warming. Therefore, researchers are searching for new catalyst materials to catalyze the HER efficiently. In this regard, a moderate ΔG_H value that is neither too high (indicative of difficult

desorption of products) or too low (indicative of difficult adsorption of reactants) is indicative of a highly active HER catalyst. Thus, we would like to develop a machine learning model that can predict the ΔG_H value as a function of the dopant added to the $Ni_2P(0001)$ surface. The dataset under consideration has been obtained using quantum-mechanical density functional theory (DFT) calculations and has been adapted from:

Wexler, R. B.; Martirez, J. M. P.; Rappe, A. M. Chemical Pressure-Driven Enhancement of the Hydrogen Evolving Activity of Ni_2P from Nonmetal Surface Doping Interpreted via Machine Learning. *J. Am. Chem. Soc.* **2018**, *140*, 4678–4683.

The Supporting Information of this article provides the machine learning data as a ZIP file. Upon downloading the ZIP file, one can find a CSV file with the name "processed_data.csv". This file contains 28 features corresponding to 56 different doped $Ni_2P(0001)$ systems of varying composition:

Symbol	Description
n_X	Number of dopants in surface layer
d_{ij}	Distance between atoms i and j in Angstroms
θ_{ijk}	Angle between atoms i, j, and k
d_{Ni-Ni}	Average distance between surface Ni atoms
$\sigma_{d_{Ni-Ni}}$	Standard deviation of distance between surface Ni atoms
$\sigma_{\theta_{Ni-Ni-Ni}}$	Standard deviation of angle between surface Ni atoms
Perimeter	Perimeter of Ni_3-hollow site
Area	Area of Ni_3-hollow site in Angstroms cubed
q_i	Residual charge on atom i in units of the electron charge
q_{ij}	Residual charge on dopant i at site j
Z	Atomic number
m	Atomic weight in amu
r	Atomic radius
$\langle q_{Ni} \rangle$	Average residual charge on Ni
$\sigma_{q_{Ni}}$	Standard deviation of residual charge on Ni
$\langle q_X \rangle$	Average residual charge on dopant X
σ_{q_X}	Standard deviation of residual charge on dopant X

4.2 Training a simple linear regression model and cross validating it

Using our previous Python and Scikit-learn experience, we first train a linear regression model with a 75:25 split for the training and test sets. To this end, we load the dataset as follows:

```
In [1]: import pandas as pd
```

```
In [2]: data=pd.read_excel('/Users/ananthgr/Documents/Acads/IISc/Courses/CH 251/
```

Next, we drop the appropriate columns to obtain the feature matrix X_{lin} and the target variable vector y:

```
In [4]: X=data.drop(columns=['dGH','Unnamed: 0'])
```

```
In [5]: y=data['dGH']
```

```
In [6]: X_lin = X.drop(columns=['X'])
```

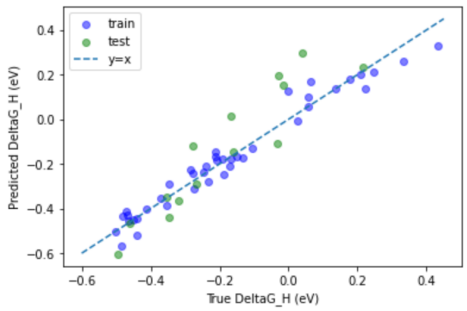
Training of a simple linear model on the training set can be done as before:

```
import numpy as np
import sklearn
from sklearn.linear_model import LinearRegression
X1_train, X1_test, y1_train, y1_test=sklearn.model_selection.train_test_split(X_lin,y,test_size=0.25,random_state=9)
reg1 = LinearRegression().fit(X1_train, y1_train)
reg1.score(X1_train,y1_train)
y1_train_pred=reg1.predict(X1_train)
y1_test_pred=reg1.predict(X1_test)
print("Test RMSE:",np.sqrt(sklearn.metrics.mean_squared_error(y1_train,y1_train_pred)))
```

```
import matplotlib.pyplot as plt
plt.scatter(y1_train,y1_train_pred,c="blue",alpha=0.5)
plt.scatter(y1_test,y1_test_pred,c="green",alpha=0.5)
plt.xlabel('True DeltaG_H (eV)')
plt.ylabel('Predicted DeltaG_H (eV)')
parity_x = np.linspace(-0.6,0.45,100)
plt.plot(parity_x,parity_x,'--')
plt.legend(["train","test","y=x"])

train_mae = sklearn.metrics.mean_absolute_error(y1_train,y1_train_pred)
train_r2=sklearn.metrics.r2_score(y1_train,y1_train_pred)
test_mae=sklearn.metrics.mean_absolute_error(y1_test,y1_test_pred)
test_r2=sklearn.metrics.r2_score(y1_test,y1_test_pred)

print('Train MAE = ', train_mae)
print('Train R^2 = ', train_r2)
print('Test MAE  = ', test_mae)
print('Test R^2  = ', test_r2)
```

We obtain the results given below:

```
Test RMSE: 0.050524681430249116
Train MAE =  0.039474276004596416
Train R^2 =  0.9588781968622488
Test MAE  =  0.09867041588478795
Test R^2  =  0.5731186132934876
```



We see that the R^2 score on the test set is not as good as one would expect for a high-quality ML model but the MAE values seem reasonable. One can also check if the model is robust across various choices of the training set by using 5-fold cross validation. This is accomplished using the KFold and cross_val_score modules in sklearn.model_selection, as follows:

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, KFold

X1_train, X1_test, y1_train, y1_test=sklearn.model_selection.train_test_split(X_lin,y,test_size=0.25,random_state=2)

# Define the linear regression model
lr_model = LinearRegression()

# Define the 5-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=6)

# Perform cross-validation and get the scores
scores = cross_val_score(lr_model, X1_train, y1_train, cv=kf, scoring='neg_mean_squared_error')

# Convert the scores to positive mean squared error (MSE)
mse_scores = -scores

# Print the MSE scores for each fold and the average MSE score
for i, mse in enumerate(mse_scores):
    print(f"Fold {i+1}: MSE = {mse:.2f}")
print(f"Average MSE: {mse_scores.mean():.2f}")
```

```
Fold 1: MSE = 0.02
Fold 2: MSE = 0.02
Fold 3: MSE = 1968.75
Fold 4: MSE = 0.03
Fold 5: MSE = 0.01
Average MSE: 393.77
```

We see that the MSE in 4 of the 5 folds is very low but something strange seems to be happening in one of the folds! This *may* occur even if you change the random_state for the train_test_split command. This implies that the model is being overfit and one needs to adopt a regularization strategy to overcome this problem.

4.3 Implementing the LASSO method and optimization of the regularization parameter

We next try the least absolute shrinkage and selection operator (LASSO) method. If you recall, one needs to choose a regularization parameter, α, that weights the linear regression coefficients. How do we decide what value of α to choose? This can also be done using 5-fold cross-validation, as shown below:

```
# Cross validation with LASSO
from sklearn.linear_model import Lasso
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.metrics import mean_absolute_error, mean_squared_error

param_grid = {
    'alpha': [0,0.1,10,100,1000],
}

lasso_regression = Lasso()
cv = KFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(lasso_regression, param_grid, cv=cv, scoring='neg_root_mean_squared_error')
grid_result = grid_search.fit(X1_train, y1_train)

# Print the results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = -1 * grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, std, params in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, std, params))

y1_train_pred=reg1.predict(X1_train)
y1_test_pred=reg1.predict(X1_test)
print("Test RMSE:",np.sqrt(sklearn.metrics.mean_squared_error(y1_train,y1_train_pred)))
```

```
Best: -0.167471 using {'alpha': 0.1}
10.612187 (20.929543) with: {'alpha': 0}
0.167471 (0.060866) with: {'alpha': 0.1}
0.232969 (0.058942) with: {'alpha': 10}
0.242063 (0.060111) with: {'alpha': 100}
0.242063 (0.060111) with: {'alpha': 1000}
Test RMSE: 0.0862111716995785
```

We see that a value of $\alpha = 0.1$ is predicted to be the best, leading to an average 5-fold-cross-validated RMSE of 0.167.

Note that $\alpha = 0$, i.e., normal linear regression again leads to a large value for the average RMSE, due to overfitting! Thus, we have successfully demonstrated the use of regularization to ensure better generalizability of the ML model using the LASSO method.