Lab ML for DS SS23

Project 2

Jan Jascha Jestel (5547158)

Mustafa Suman (5564676)

Gabriele Inciuraite (5208806)

In [86]:

```
import numpy as np
import pandas as pd

from matplotlib import pyplot as plt
import seaborn as sns

from scipy.io import loadmat
from scipy.spatial.distance import pdist
from scipy.stats import norm

from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso, LinearRegression
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedKFold
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_err
from sklearn.model_selection import ParameterGrid
from sklearn.utils import shuffle


from itertools import combinations_with_replacement, combinations
import pickle

# Set random seed
np.random.seed(42)
```

1. Importing the QM7 Dataset

In [87]:

```
# The QM7 dataset consists of 7165 organic molecules, each of which is comp
qm7 = loadmat("./qm7.mat")
```

```
print(qm7.keys())
```

```
dict_keys(['__header__', '__version__', '__globals__', 'X', 'R', 'Z', 'T', 'P
'])
```

In [88]:
```
# R (7165×23×3) contains for each molecule and atom a triplet representing
display(qm7['R'].shape)
display(qm7['R'][0])
```

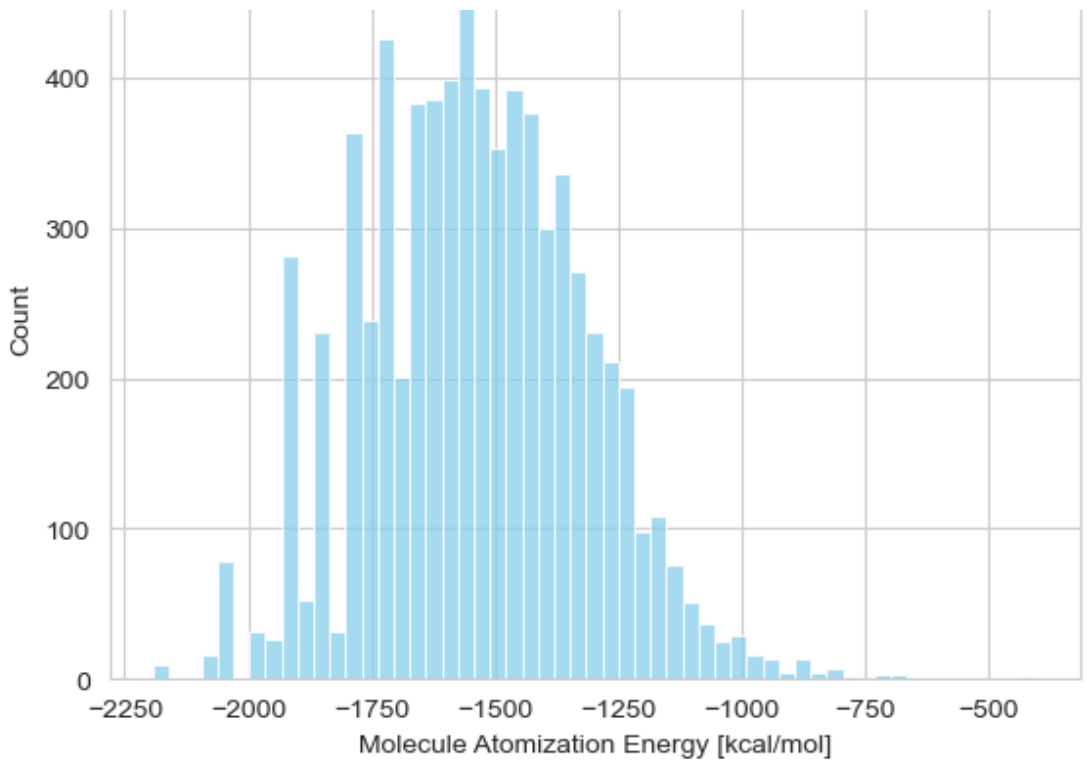
```
(7165, 23, 3)
array([[ 1.886438  , -0.00464873, -0.00823921],
       [ 3.9499245 , -0.00459203,  0.00782347],
       [ 1.1976895 ,  1.9404842 ,  0.00782347],
       [ 1.1849339 , -0.99726516,  1.6593875 ],
       [ 1.2119948 , -0.9589793 , -1.710958  ],
       [ 0.        ,  0.        ,  0.        ],
       [ 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.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ]], dtype=float32)
```

In [89]:
```
# Z (7165×23) contains for each molecule and atom of the molecule the corre

# 0 == no atom at this index
# 1 == hydrogen (H) 1
# 6 == carbon (C) 4
# 7 == nitrogen (N) 3
# 8 == oxygen (O) 2
# 16 == sulfur (S) 6
display(qm7['Z'].shape)
display(qm7['Z'][0])
atoms = ["H", "C", "N", "O", "S"]
```

```
(7165, 23)
array([6., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0.], dtype=float32)
```

In [90]:
```
# T (1 x 7165) contains for each molecule the atomization energy (computed
display(qm7['T'].shape)
display(qm7['T'][0, 0])

g = sns.histplot(qm7['T'][0], color="skyblue")
g.set_xlabel('Molecule Atomization Energy [kcal/mol]')
plt.title("Distribution of Atomization Energy")
plt.show()
```

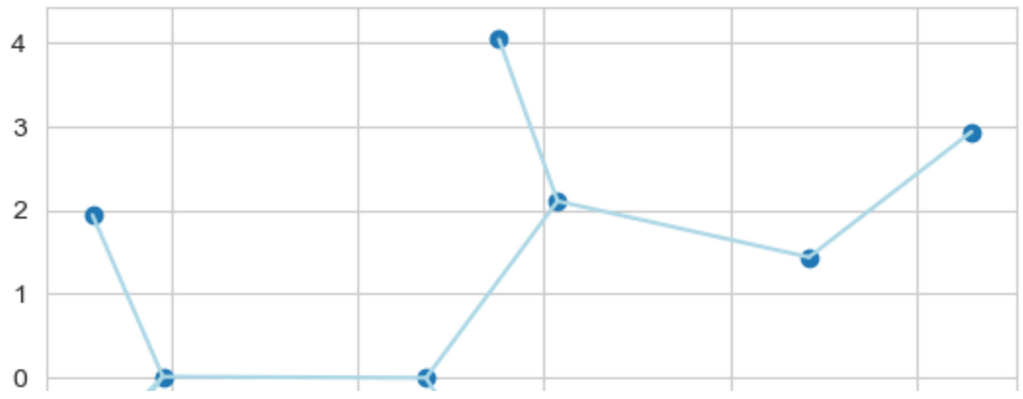
```
(1, 7165)
-417.96
```

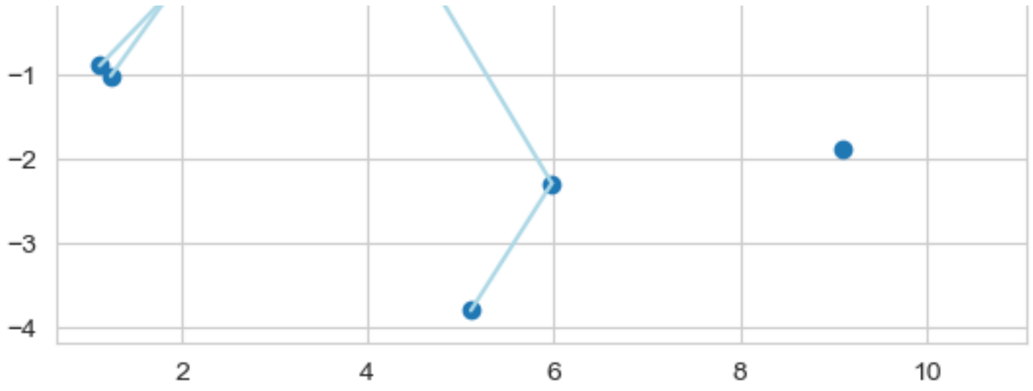


1.1 Visualizing Molecules

"Quick and dirty approach"

In [91]:
```python
nr = 6432

mol = qm7["R"][nr]
sym = qm7["Z"][nr]

sym = sym[sym > 0]
mol = mol[: len(sym)]

plt.scatter(x=mol[:, 0], y=mol[:, 1])


# generate bonds
thresh = 3

mask = np.argwhere(pdist(mol, metric="euclidean") < thresh)[:, 0]
bonds = np.vstack(np.triu_indices(len(mol), 1)).T[mask]

for i in range(len(bonds)):
    plt.plot(mol[bonds[i]][:, 0], mol[bonds[i]][:, 1], color="lightblue")
```





Observation: In this 2-D visualisation we chose a fixed threshold of three. This works well enough for many bonds, but not for all of them. Different atoms have different bond lengths: as can be seen in this particular example, one atom is further away from the others. In addition to that, some atoms have multiple bonds between the same atoms (e.g. double, tripple, etc.) which is not expressed in the visualisation.

Atomic Simulation Environment

To get a more sophisticated 3D-visualization we make use of some other libraries.

In [92]:

```
from ase import Atoms
import nglview
import ase.visualize

system = Atoms(positions=mol, symbols=sym)
#ase.visualize.view(system, viewer="x3d")

view = nglview.show_ase(system)
view.add_ball_and_stick() # how to specify bonds?
view

# Other viz approaches
#https://www.kaggle.com/code/mykolazotko/3d-visualization-of-molecules-with
```

```
NGLWidget()
```

2. Simple atom-based Data Representation

We first decomposed the molecules into individual elements - atoms - and saved a representation of each molecule as an array with the counts of each atom type.

In [93]:

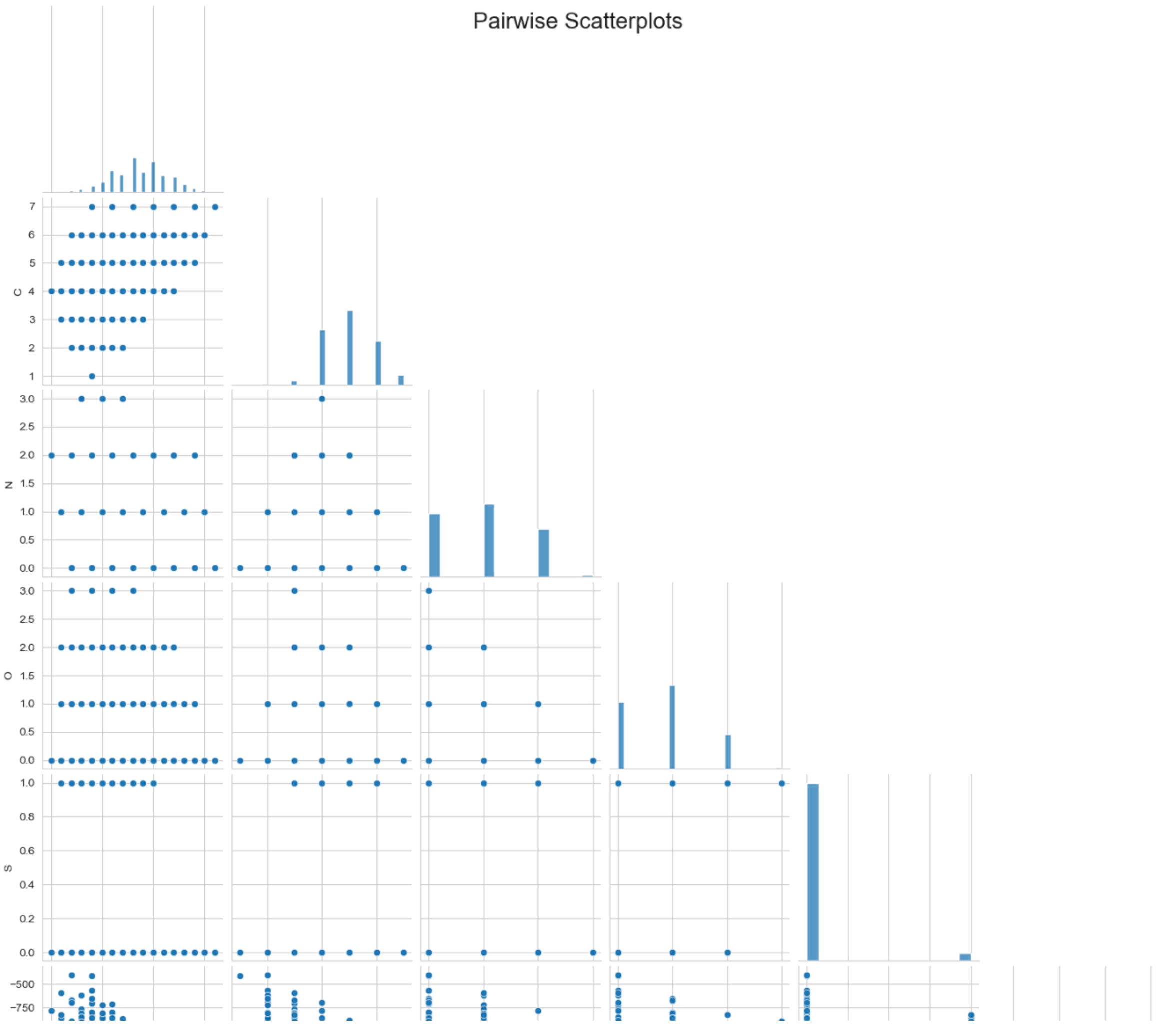
```
# one hot encoding of [HCNOS]

# 1 == hydrogen (H)
# 6 == carbon (C)
# 7 == nitrogen (N)
# 8 == oxygen (O)
# 16 == sulfur (S)

z = qm7["Z"].astype(np.int8)

z[z == 1] = 1
z[z == 6] = 2
z[z == 7] = 3
z[z == 8] = 4
```

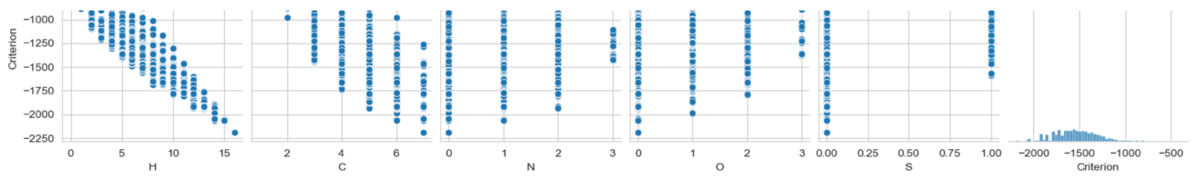
```
z[z == 16] = 5

Z_hot = np.eye(6)[z]
Z_hot = Z_hot[:, :, 1:]  # drop 0 dimension

# sum of to create the representation
x = np.sum(Z_hot, axis=1).astype(int)

print("Shape:", x.shape)
print("Resulting representations:")
print(x)
```

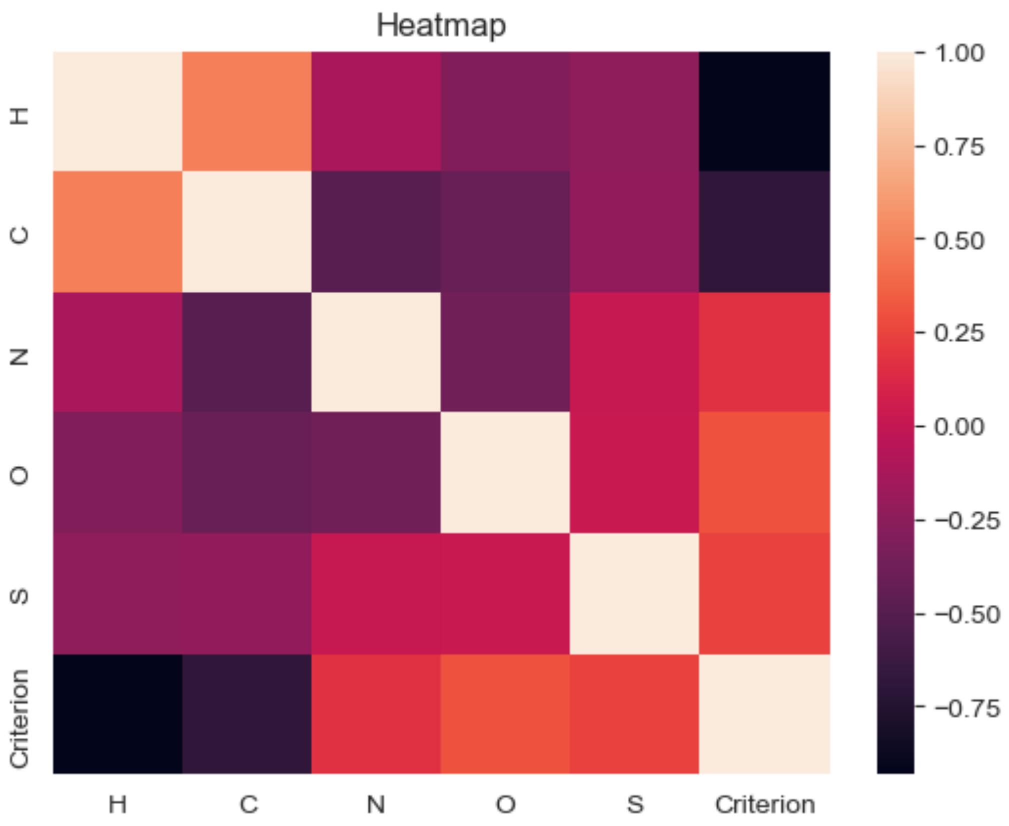
```
Shape: (7165, 5)
Resulting representations:
[[ 4  1  0  0  0]
 [ 6  2  0  0  0]
 [ 4  2  0  0  0]
 ...
 [ 9  6  1  0  0]
 [10  7  0  0  0]
 [12  7  0  0  0]]
```

In [94]:

```
y = np.reshape(qm7["T"][0], (-1, 1))
df = pd.DataFrame(np.concatenate((x, y), axis=1), columns = ["H", "C", "N",

sns.pairplot(df, corner=True)
plt.suptitle("Pairwise Scatterplots", fontsize=20)
plt.show()

sns.heatmap(df.corr())
plt.title("Heatmap")
plt.show()
```





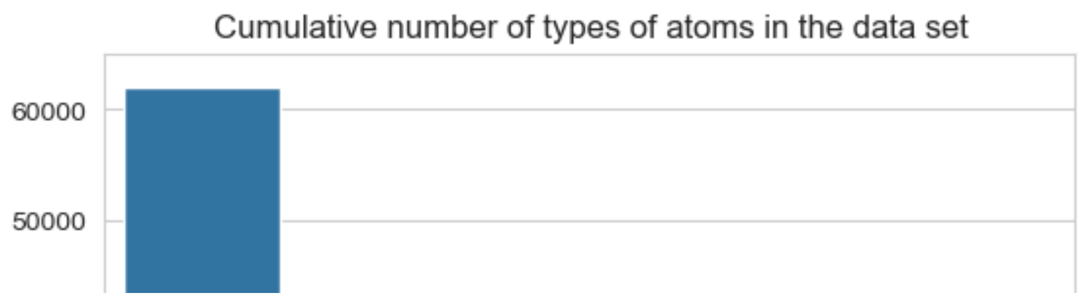


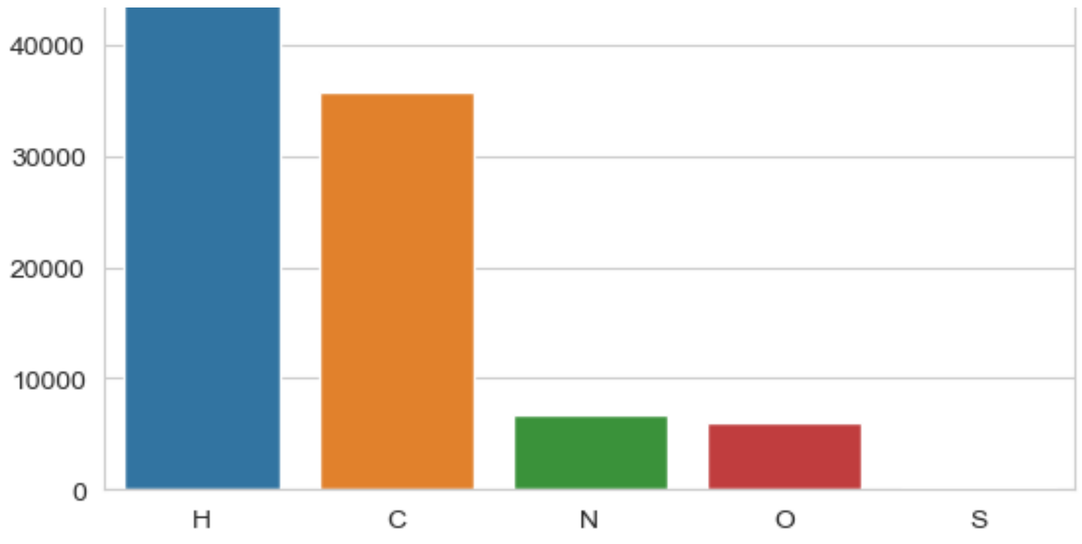
In [95]:
```python
print(df.corr())
```

```
                  H         C         N         O         S  Criterion
H          1.000000  0.483864 -0.125100 -0.298140 -0.236983  -0.931799
C          0.483864  1.000000 -0.486695 -0.413210 -0.225661  -0.688390
N         -0.125100 -0.486695  1.000000 -0.373496  0.017874   0.169763
O         -0.298140 -0.413210 -0.373496  1.000000  0.025079   0.299107
S         -0.236983 -0.225661  0.017874  0.025079  1.000000   0.243362
Criterion -0.931799 -0.688390  0.169763  0.299107  0.243362   1.000000
```

Observation: The pairwise scatterplots and correlation matrix reveal a lack of multicollinearity in the data - the predictor variables are not highly correlated with one another, except for hydrogen and carbon (corr = 0.48). We can also see a strong linear relationship between the atomisation energy and hydrogen (corr = -0.93) as well as carbon (corr = -0.69).

In [96]:
```python
g = sns.barplot(x=atoms, y=np.sum(x, axis=0))
g.set_title('Cumulative number of types of atoms in the data set')
plt.show()
```





Observation: It is worth noting that hydrogen and carbon are the most common atoms in the data set.

2.1 ML Model: Ridge Regression

We then split the data into train and test subsets and centered numbers of the atoms in the molecules.

In [97]:
```
def split_and_center(X, y, test_size, shuffle=True):
    # split train / test
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=tes

    # center data and target
    X_train_mean = np.mean(X_train, axis=0)
    y_train_mean = np.mean(y_train)
    print(X_train_mean)

    X_train = X_train - X_train_mean
    X_test = X_test - X_train_mean

    y_train = y_train - y_train_mean
    y_test = y_test - y_train_mean
    #y_train = np.sqrt(y_train)
    #y_test = np.sqrt(y_test)
    print(y_test)

    return X_train, X_test, y_train, y_test



X_train, X_test, y_train, y_test = split_and_center(x, qm7["T"][0], test_si
```

```
[8.61894317 4.98703888 0.92961117 0.83808574 0.04287139]
[-373.02234     53.44763    231.3977    ...  172.48767   -513.9824
   -1.2923584]
```

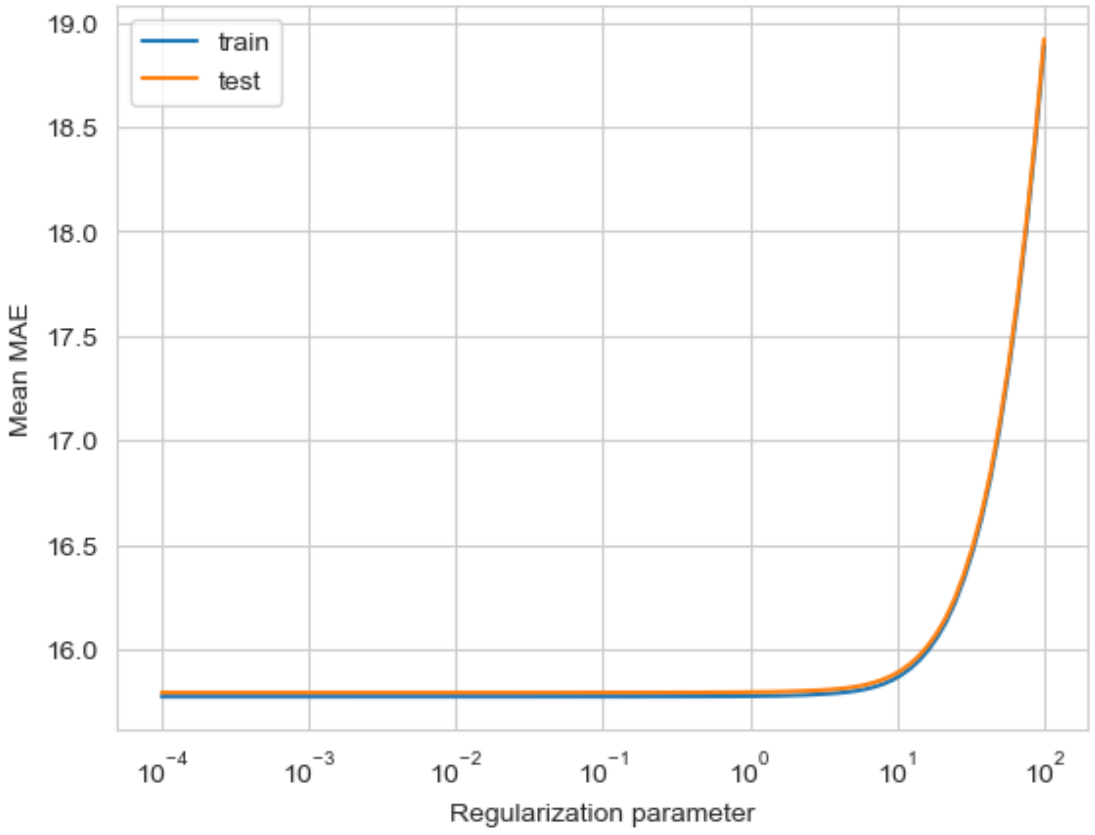
In order to tune the regularisation parameter alpha for the Ridge regression model, we performed a grid search on the training data.

In [98]:
```
# Grid search to tune alpha using 10-fold cross validation

ridge = Ridge()
```

```
alpha_values = np.logspace(np.log10(1e-4), np.log10(100), num=100)
grid = dict(alpha=alpha_values)
search = GridSearchCV(
    ridge,
    grid,
    scoring=("neg_mean_absolute_error"), # 'r2', 'neg_mean_squared_error'
    cv=10,
    n_jobs=-1,
    refit="neg_mean_absolute_error",
    return_train_score=True,
)

results = search.fit(X_train, y_train)
```

In [99]:

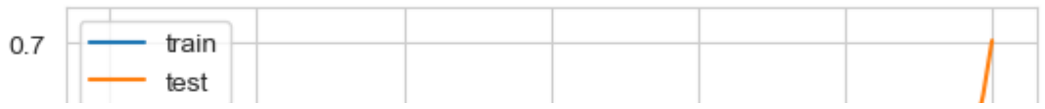
```
ax = sns.lineplot(x=alpha_values, y = -results.cv_results_['mean_train_scor
ax = sns.lineplot(x=alpha_values, y = -results.cv_results_['mean_test_score
ax.set_xscale('log')
ax.set_xlabel('Regularization parameter')
ax.set_ylabel('Mean MAE')
plt.legend(loc='upper left')
plt.show()
```

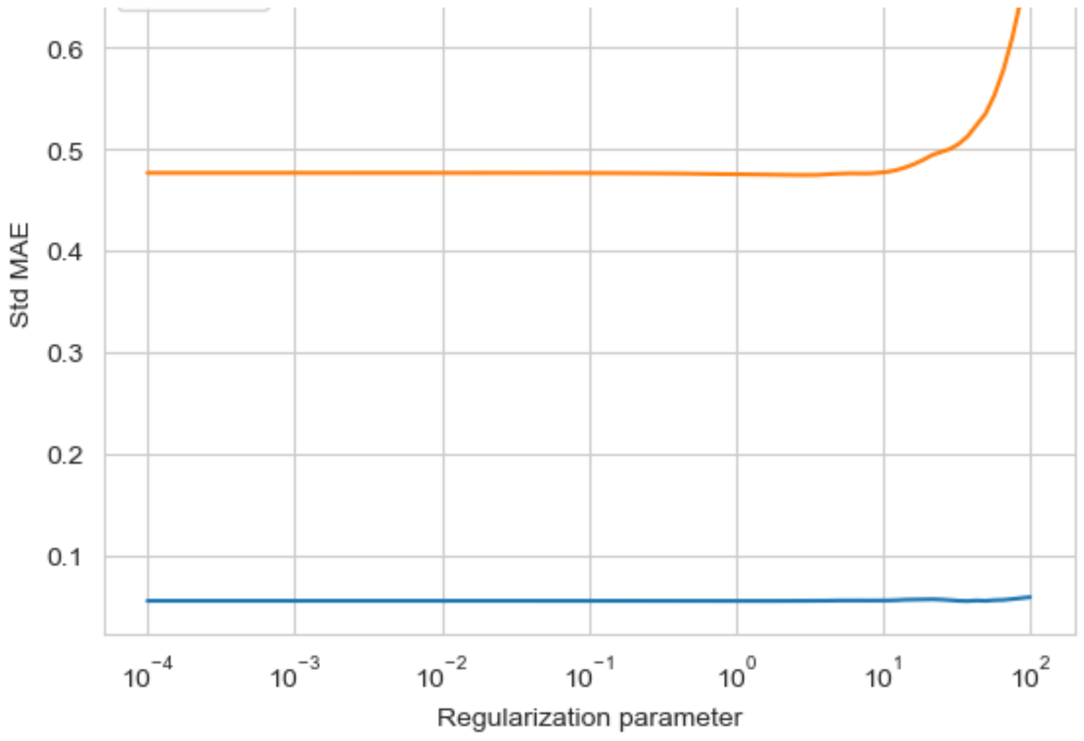


Observation: Low regualarisation parameter achieves the lowest MAE.

In [100…

```
ax = sns.lineplot(x=alpha_values, y = results.cv_results_['std_train_score'
ax = sns.lineplot(x=alpha_values, y = results.cv_results_['std_test_score']
ax.set_xscale('log')
ax.set_xlabel('Regularization parameter')
ax.set_ylabel('Std MAE')
plt.legend(loc='upper left')
plt.show()
```





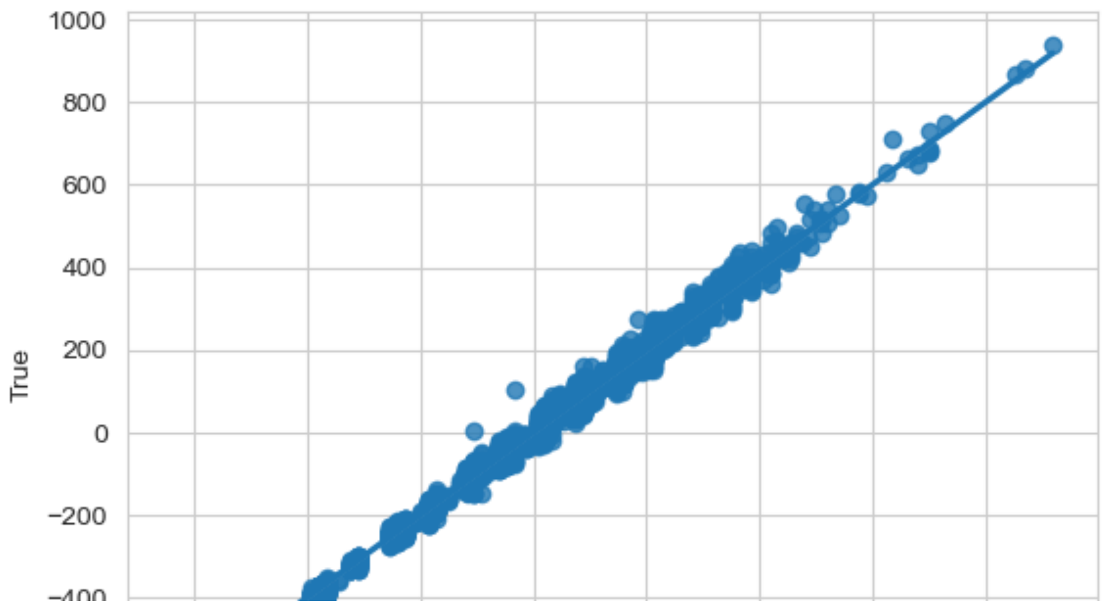
Observation: The standard deviation of the absolute error is very low and stable for low regualarisation parameter.

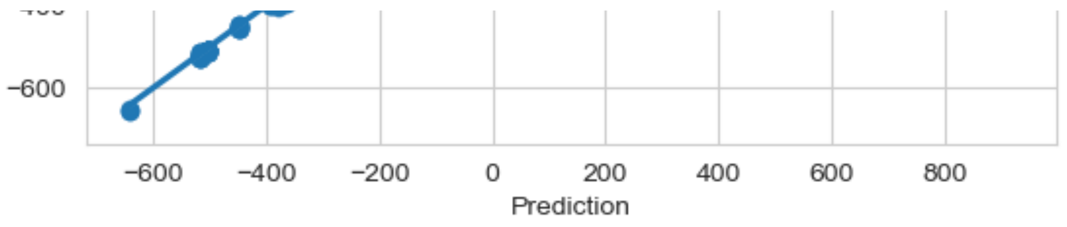
We plotted the regression model using the very low regularisation parameter (alpha= 0.0001) which resulted in our parameter tuning.

In [101…

```
y_pred = search.best_estimator_.predict(X_test)
print("Best parameter:", results.best_params_)
print(f"R2: {r2_score(y_test, y_pred):.3f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred):.3f}")
print(f"MSE: {mean_squared_error(y_test, y_pred):.3f}")
g = sns.regplot(x=y_pred, y=y_test)
g.set_xlabel("Prediction")
g.set_ylabel("True")
plt.show()
```

```
Best parameter: {'alpha': 0.0001}
R2: 0.992
MAE: 15.461
MSE: 403.582
```



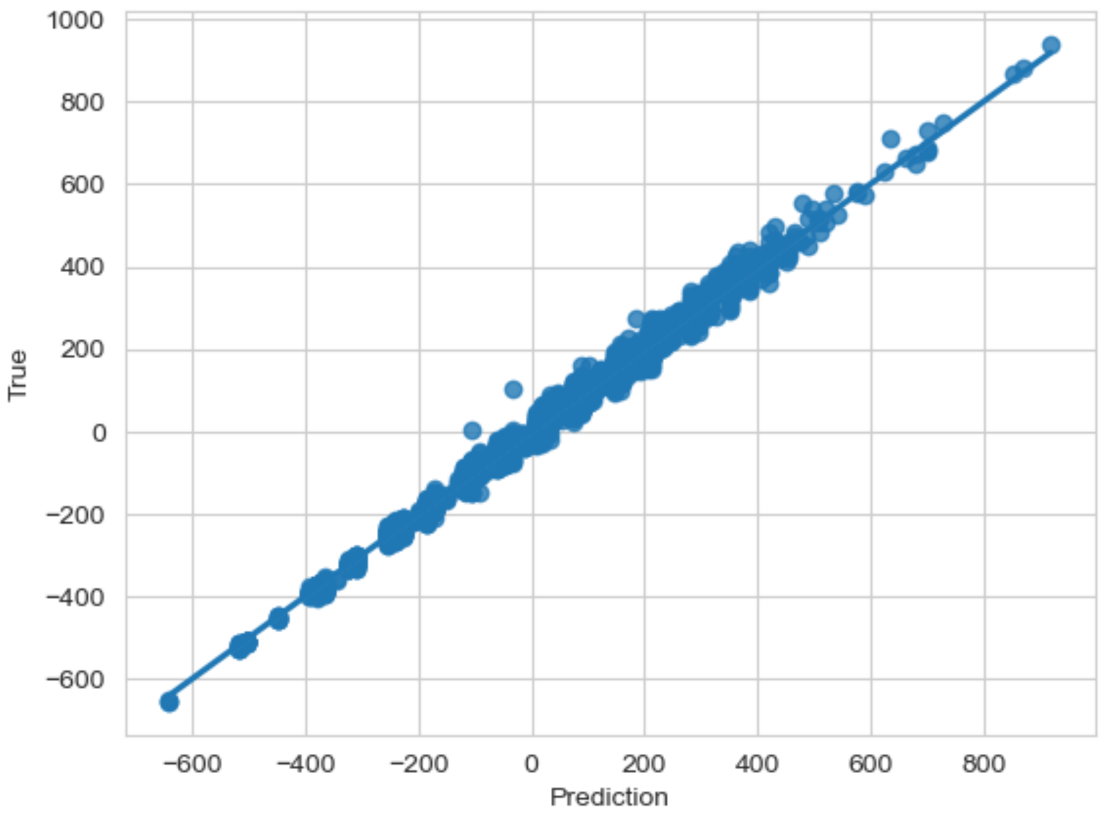


Since the regularisation parameter is so small we compared the results with performing a multiple linear regression without regularisation.

In [102…

```
lr = LinearRegression().fit(X_train, y_train)
y_pred_lr = lr.predict(X_test)
print(f"R2: {r2_score(y_test, y_pred_lr):.3f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred_lr):.3f}")
print(f"MSE: {mean_squared_error(y_test, y_pred_lr):.3f}")

g = sns.regplot(x=y_pred_lr, y=y_test)
g.set_xlabel("Prediction")
g.set_ylabel("True")
plt.show()
```

```
R2: 0.992
MAE: 15.461
MSE: 403.582
```



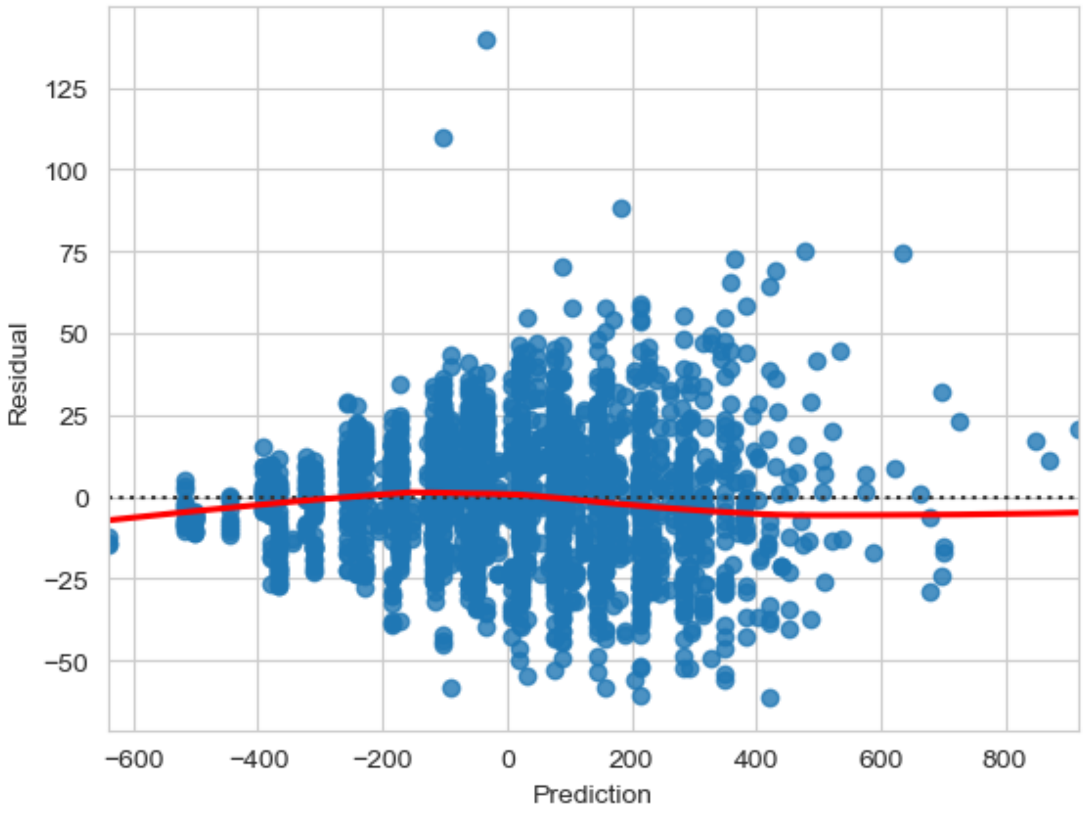
Observations: The explained variance and error scores for these two regression models are identical. This is due to the fact that in the Ridge regression the penalty parameter is effectively 0, basically resulting in unpenalised linear regression model. Taken together with the tendency for the Ridge model to perform increasingly worse with higher regularisation term, we conclude that the Ridge regression introduces a non-beneficial bias. The possible reduction of the variance of the estimator does not seem to recuperate/improve the overall performance. We can attribute this to fackt that we did not observe high multicolinearity in the data. In the absence of multicolinearity, linear regression tends to perform equally well or even better than Ridge regression. Ridge

Regression, on the other hand, can come in hand in the case of a high number of included covariates.

We also observed that the explained variance is very high, even when fitting the regression to a low number of data points. This, too, points to a strong linear relationship between the independent and dependent variables. We observed that the most common atoms in the data, hydrogen and carbon, had a high linear relationship with the criterion. We therefore conclude that the underlying relationship in the data is highly linear.

In [103…

```
g = sns.residplot(x=y_pred, y=y_test, lowess=True, line_kws=dict(color="r"))
g.set_xlabel("Prediction")
g.set_ylabel("Residual")
plt.show()
```



Observation: The LOWESS curve, even if not completely straight, is not showing a strong trend in the residual plot. However, there is more variance in the residuals of the positive predictions. This indicares heteroscedasticity in the data (the STDs of a predicted variable, monitored over different values of independent variables, are non-constant). This might point to biased standard errors, but should not affect the estimation of the regression coefficients, which explain 99.2 % of the variance in atomisation energy given our data set.

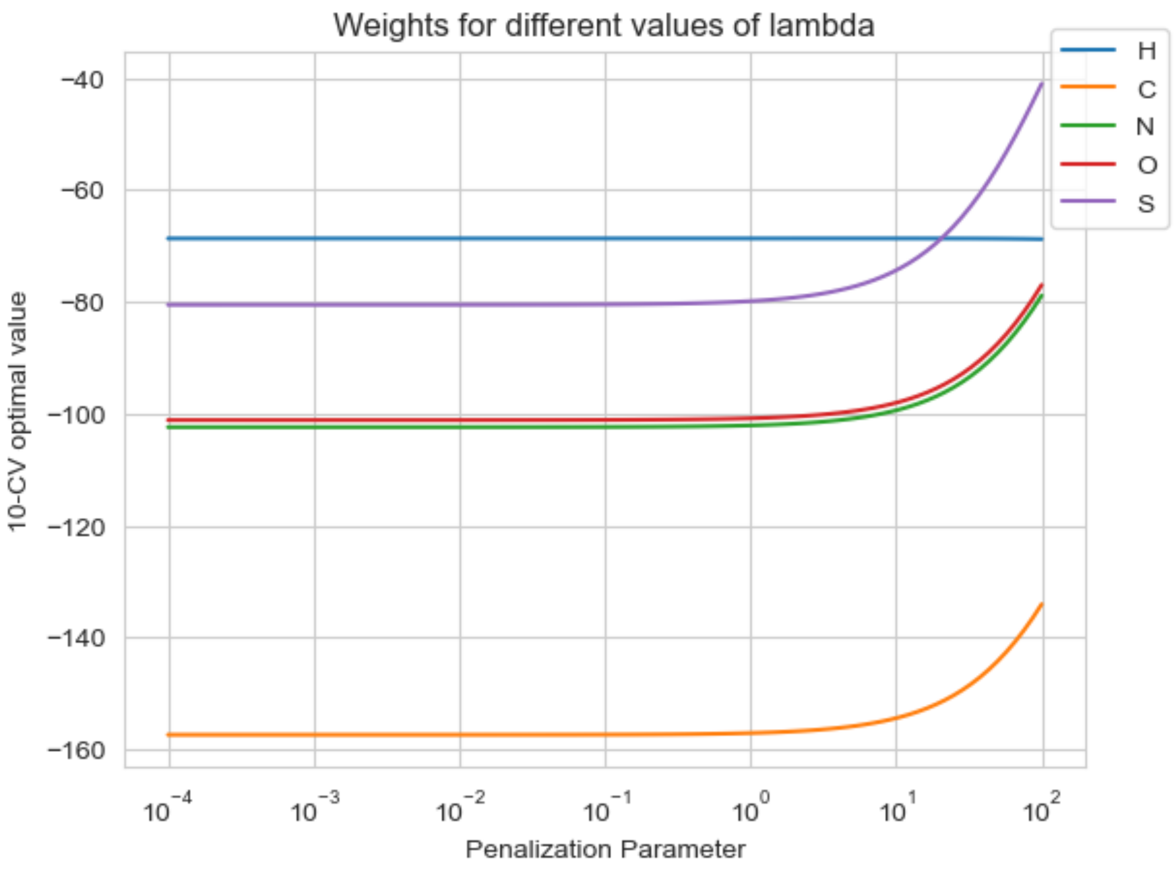
Visualization for intuition: Impact of penalization on weights

In [104…

```
ridge_ = Ridge()

weights_ = []
scores_ = []
for alpha in alpha_values:
    grid_ = dict(alpha = [alpha])
    search_ = GridSearchCV(ridge_, grid_, scoring='neg_mean_squared_error
    results_ = search_.fit(X_train, y_train)
    weights_ += [results_.best_estimator_.coef_]
    scores_ += [results_.best_score_]

weights_ = np.array(weights_)
```

In [105…

```
plt.plot(alpha_values, weights_)
plt.legend(atoms, bbox_to_anchor=(1.1, 1.05))
plt.xscale("log")
plt.title("Weights for different values of lambda")
plt.ylabel("10-CV optimal value")
plt.xlabel("Penalization Parameter")
plt.show()
```



Intuitively, we can see how the penalization works. Ridge introduces a shrinkage towards 0 with an increasing value of the penalization paramter. This leads to an introduced bias which possibely can reduce the estimators performance (i.e. MSE) by reducing the variance of the estimator.

2.2 Explanations: Simple atom-based Representation

Fitting our best found model, we get the following MAE and weights for the different atom Types:

In [106…

```
print(f"Best performing regularization parameter: {results.best_params_}")
print(f"MAE on validation set: {mean_absolute_error(y_test, search.best_est
print("Weights Ri for H, C, N, O, S:", search.best_estimator_.coef_)
```

```
Best performing regularization parameter: {'alpha': 0.0001}
MAE on validation set: 15.461 kcal/mol
Weights Ri for H, C, N, O, S: [ -68.68727583 -157.410442   -102.41624528 -101
.15629173  -80.54594189]
```

In the following, we will compare our gained insights with existing chemical knowledge and literature.

Recap: As stated in the exercise, The atomization energy of a molecule is the energy generated by dissociating all atoms from the molecule, i.e. moving atoms far apart so that the bonds between atoms are broken. Because it consumes energy to break these bonds, the atomization energy is typically a negative quantity.

The atomization energy is a complex measure influenced by a variety of factors such as the incorporated bond strengths & lengths, molecular size, bond types, electronic configurations, intermolecular interactions and the molecular geometry. Related measures are the ionisation energy or electron affinity.

For the sake of simplicity, one can make use of the additivity property of the atomization energy of a molecule to get an estimate, i.e. add up the energy needed to break up the atom pairs within a molecule cf. Usefully, there are several tables of empirically determined values for different pairs of atoms. To double check our estimates, we will make use of the one found in [Neufingerl: Chemie 1 - Allgemeine und anorganische Chemie, Jugend & Volk, Wien 2006; ISBN 978-3-7100-1184-9. S. 47].

The bounding energy of pairs of atoms depend on the bond length, the polarity of the bond and the type of the bond (simple, double, triple, ...). The more bonds are prevalent within a pair of atoms, the harder it is to separate them. In the table below, if there are multiple values given it indicates the energy needed for different types of bonds. For further calculations, the equally weighted average is taken since we have no further knowledge in the dataset about the bond types given. It is important to note that this approach serves as a simplified orientation.

In [107…

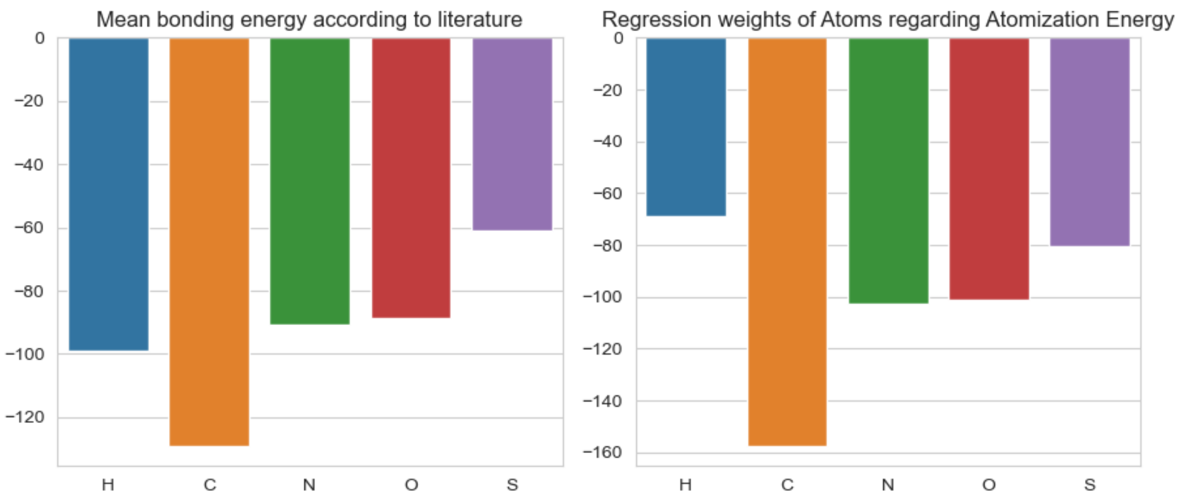
```python
H = [436, 413, 391, 463, 367]
C = [np.nan, (348 + 614 + 839)/3, (305 + 615 + 891)/3, (358 + 745)/2, (272 
N = [np.nan, np.nan, (163 + 418 + 945)/3, (201 + 607)/2, 225]
O = [np.nan, np.nan, np.nan, (146 + 498)/2, 420] # last entry: bond-type = 
S = [np.nan, np.nan, np.nan, np.nan, 255]

binding_energy_pairs = np.array([H, C, N, O, S])
print("Binding Energy table; order: (H,C,N,O,S):\n")
print(binding_energy_pairs)
mean_binding_energies = np.nanmean(binding_energy_pairs, axis=1)
mean_binding_energies *= 0.239 # convert kJ to kcal
```

```
Binding Energy table; order: (H,C,N,O,S):

[[436.         413.         391.         463.         367.        ]
 [         nan 600.33333333 603.66666667 551.5        404.        ]
 [         nan          nan 508.66666667 404.        225.        ]
 [         nan          nan          nan 322.        420.        ]
 [         nan          nan          nan          nan 255.        ]]
```

In [108…

```python
fig, axs = plt.subplots(1,2, figsize=(9,4))

g = sns.barplot(x=atoms, y=(-1) * mean_binding_energies, ax=axs[0])
g.set_title("Mean bonding energy according to literature")
```

```
f = sns.barplot(x=atoms, y=search.best_estimator_.coef_, ax=axs[1])
f.set_title("Regression weights of Atoms regarding Atomization Energy")
plt.tight_layout()
plt.show()
```

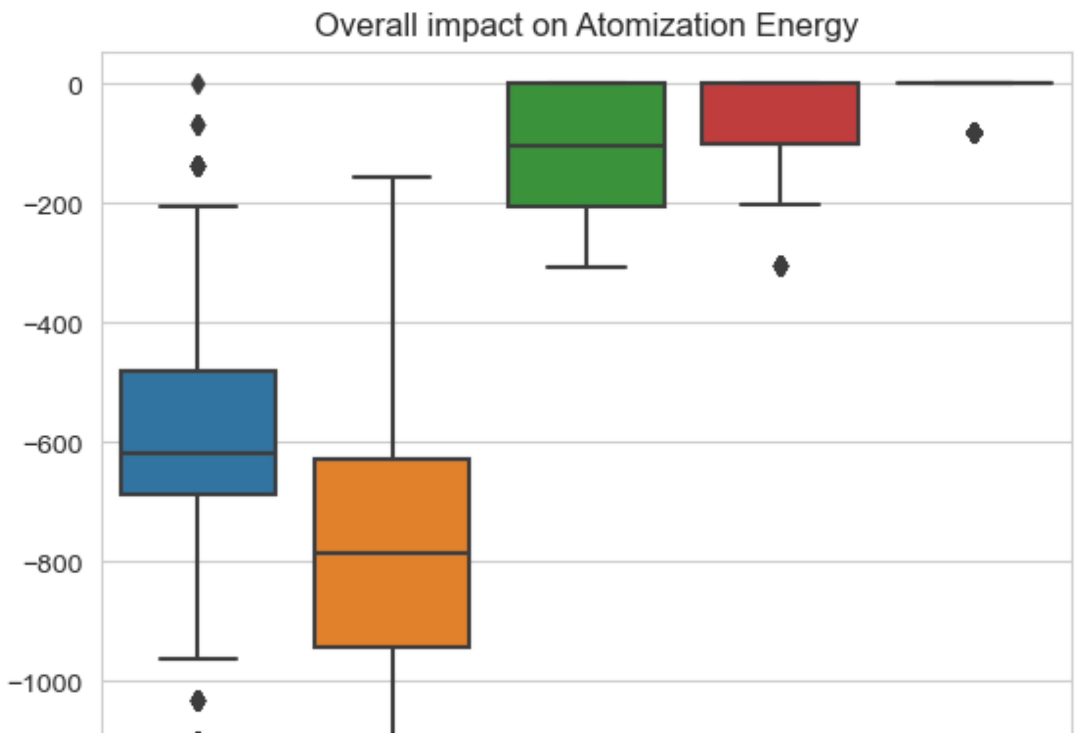


Comparing our results with the above-described literature, we can see that our results nearly perfectly match the findings. This is true for the ordering as well as the magnitudes of the atom influences. The only exception here is the hydrogen atom which's impact is overestimated. Also, the Carbon impact is a little overestimated as well. However, as emphasized we are dealing with simplified approaches and differences may reasonably occur. One possible reason for this might be the pairwise discoverd correlations among H, C and the target variable.

Overall, the results seem surprisingly satisfying!

In [109…

```
#fig = plt.figure(figsize=(10,10))
g = sns.boxplot(x * search.best_estimator_.coef_)
g.set_title("Overall impact on Atomization Energy")
g.set_xticklabels(atoms)
plt.show()
```



3. Pairs-of-atoms-based Data Representation

To take the mutual distances between atoms into consideration, we decomposed each molecule into its set of pairs of atoms. To do that, we generated a one-hot encoding of distances by binning them into multiple intervals. We inspected tables with lengths of different atom bonds, such as found here: http://www.pathwaystochemistry.com/wp-content/uploads/BondLengths.jpg. Based on this information, we assume that the shortest bond given our atom pairs must be HH with 1.4 a.u. and the longest - SO with 5.0 a.u. We therefore set the range of distances to (1.3, 5.1) in this data representation.

To avoid introducing unnatural discontinuities into the model, we enabled not only a hard indicator function, but also a soft indicator function (Gaussian function with mean at the center of the interval and fixed variance).

In [110…

```
# create dictionary to translate atom combinations to number between 0 and
d = {c: i for i, c in enumerate(combinations_with_replacement([1, 2, 3, 4,

# add inverse tuples pointing to same number
inv = {}
for c in d:
    inv[c[1], c[0]] = d[c]

# merge dictionaries
d = d | inv
del inv
```

In [111…

```
# generate arrays with the distances and the types of all pairs
N_COMB = 824783
pair_dist = np.zeros(N_COMB) # all distances
pair_type = np.zeros(N_COMB).astype(int) # all types
mol_split = np.zeros(7165 + 1).astype(int) # indices where to split the arr
pair_idx = 0

for i in range(len(z)):
    sym = z[i]
    sym = sym[sym != 0]
    mol = qm7["R"][i][: len(sym)]

    dist = pdist(mol, metric="euclidean")
    comb = [*combinations(sym, r=2)]
    n_combs = len(comb)

    pair_dist[pair_idx : pair_idx + n_combs] = dist
    pair_type[pair_idx : pair_idx + n_combs] = [d[x] for x in comb]

    mol_split[i] = pair_idx
    pair_idx += n_combs

mol_split[-1] = N_COMB
```

In [112…

```
def generate_representaions(THETA_1, THETA_M, STD, M, soft = True):
    intervals = np.linspace(THETA_1, THETA_M, M)
    interval_size = (THETA_M - THETA_1) / (M - 1)
    interval_centeres = intervals[:-2] + interval_size

    # generate phi_A(Ei) using specified hard or soft encoding
    phi_A = np.zeros((len(pair_dist), M))

    if soft:
        for j, mu in enumerate(interval_centeres):
            phi_A[:, j] = norm.pdf(pair_dist, loc=mu, scale=STD)
    else:
        indices = np.floor((pair_dist - THETA_1) / interval_size).astype(in
        phi_A[indices < M] = np.eye(M)[indices[indices < M]]

    # generate phi_B(Ei)
    phi_B = np.eye(15)[pair_type].astype(int)

    # generate phi(Ei)
    phi_AB = phi_A[:, :, None] * phi_B[:, None, :]
    phi_AB = phi_AB.reshape(N_COMB, -1)

    # aggregate for molecule representation
    reps = np.zeros((len(z), M * 15))
    for i in range(len(mol_split)):
        reps[i - 1, :] = np.sum(phi_AB[mol_split[i - 1] : mol_split[i]], ax

    return reps
```

3.1 ML Models: Pairs-of-atoms-based Data Representation

In order to perform a “sanity check” on the model, we again utilised the typical bond lengths. For example, for different number of bonds between two carbon atoms, we expect high coefficient values at around 2.3, 2.5, and 2.9. Here is a table including the bond lengths for the pairs of atoms in our data representation, based on our research:

HH: 1.40, HC: 2.08 HN: 1.85 HO: 1.78 HS: 2.49 CC: 2.27, 2.53, 2.91 CN: 2.17, 2.40, 2.78 CO: 2.14, 2.29, 2.70 CS: 3.42 NN: 2.14, 2.31, 2.65 NO: 2.57 NS: 1.50 OO: 2.29, 2.49 OS: 2.51, 5.01 SS: 3.93

In [113…

```
def plot_pp(model, M, THETA_1, THETA_M, **kwargs):

    # All pairwise potentials plots
    fig, axes = plt.subplots(5, 3, figsize=(12, 20))
    axes = axes.flatten()
    pair_names = [*combinations_with_replacement(atoms, r=2)]
    intervals = np.linspace(THETA_1, THETA_M, M)

    bond_lenths = [[1.40],
                   [2.08],
                   [1.85],
                   [1.78],
                   [2.49],
                   [2.27, 2.53, 2.91],
                   [2.17, 2.40, 2.78],
                   [2.14, 2.29, 2.70],
                   [3.42],
                   [2.14, 2.31, 2.65],
                   [2.57],
                    [1.50],
                    [2.29, 2.49],
                    [2.51, 5.01],
                    [3.93]]

    for i, bonds in enumerate(bond_lenths):
        g = sns.lineplot(x=intervals, y=model.coef_.reshape(M, 15)[:, i], a
        for bond in bonds:
            axes[i].axvline(x = bond, color="green", linestyle='dashed')
        g.set_xlabel("Distance")
        g.set_ylabel("Coefficient")
        name = str(pair_names[i]).replace("'", "")
        g.set_title(f"Pairwise potentials {name}")
    plt.tight_layout()
```

In [114…

```
def fit_and_eval_model(Model, ALPHA, X_train, y_train, X_test, y_test,  M, '
    model = Model(alpha=ALPHA, max_iter= 10000)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    print(f"R2: {r2_score(y_test, y_pred):.3f}")
    print(f"MAE: {mean_absolute_error(y_test, y_pred):.3f}")
    print(f"MSE: {mean_squared_error(y_test, y_pred):.3f}")
    print(f"#coef = 0: {np.sum(model.coef_ == 0)}")
    print(f"#coef < 1e-10: {np.sum(model.coef_ < 1e-10)}")

    plot_pp(model, M, THETA_1, THETA_M)
```

GRID SEARCH TUNING (soft encoding)

We computed a grid search in order to tune the regularization parameter λ, the size of intervals M and the SD of the Gaussian function and therefore minimise the error on validation data.

It is worth noting that we have observed a shuffling effect in the data. We therefore shuffle the data once and use the same order for every grid search. We can only speculate if the molecules have some kind of ordering and more complex ones are contained in the last 30 % of the data set. Or maybe it's purely a coincidence. It is worth noticing, that we also can not be sure how representative the sample is, given how many complex molecules are possible.

In [115…

```
param_grid = {
    "M": [10, 40, 70, 100],
    "STD": np.linspace(0.05, 0.15, 10),
    "THETA_1": [1.3], #[2],
    "THETA_M": [5.1] #[4, 8],
}

print(len(list(ParameterGrid(param_grid))))


alpha_values = np.logspace(np.log10(1e-4), np.log10(10000), num=100)

results = []

# use same shuffle for every grid search but shuffle once because it seems
shuffled_index = shuffle(range(len(qm7["T"][0])))
```

```
# Do search or load results from grid_search.p
DO_SEARCH = False

if DO_SEARCH:
    for p in ParameterGrid(param_grid):

        reps = generate_representaions(**p, soft=True)

        # Use 80 % of the data for grid search (report real test error addi
        X_train, X_test, y_train, y_test = split_and_center(
            reps[shuffled_index],
            qm7["T"][0][shuffled_index],
            test_size=0.2,
            shuffle=False,
        )

        ridge = Ridge()
        grid = dict(alpha=alpha_values)
        search = GridSearchCV(
            ridge,
            grid,
            scoring=("neg_mean_absolute_error"),
            cv=5,
            n_jobs=-1,
            # refit="neg_mean_absolute_error",
            return_train_score=True,
        )
        search = search.fit(X_train, y_train)

        #print(p, search.best_score_, search.best_params_)

        results += [{"params": p, "search": search}]
    pickle.dump(results, open("grid_search.p", "wb"))
else:
    results = pickle.load(open("grid_search.p", "rb"))
```

```
40
```

In [116…

```
res_df = pd.DataFrame([r["params"] for r in results])
res_df["model_idx"] = res_df.index
res_df['best_score'] = [r['search'].best_score_ for r in results]
res_df['best_alpha'] = [r["search"].best_params_["alpha"] for r in results]
res_df['n_coef_smaller_1e-4'] = [np.sum(r['search'].best_estimator_.coef_ <

# Print results with best MAE
res_df.sort_values(by='best_score', ascending=False)
```

Out[116…

	M	STD	THETA_1	THETA_M	model_idx	best_score	best_alpha	n_coef_
30	100	0.050000	1.3	5.1	30	-4.492457	0.298365	
31	100	0.061111	1.3	5.1	31	-4.701077	0.359381	
20	70	0.050000	1.3	5.1	20	-4.748572	1.917910	
21	70	0.061111	1.3	5.1	21	-4.769684	1.097499	
32	100	0.072222	1.3	5.1	32	-4.820479	1.097499	
22	70	0.072222	1.3	5.1	22	-4.827104	0.756463	
33	100	0.083333	1.3	5.1	33	-4.884773	0.247708	
23	70	0.083333	1.3	5.1	23	-4.886615	0.170735	
34	100	0.094444	1.3	5.1	34	-4.948917	0.067342	

24	70	0.094444	1.3	5.1	24	-4.951265	0.046416
35	100	0.105556	1.3	5.1	35	-5.027684	0.038535
25	70	0.105556	1.3	5.1	25	-5.032327	0.026561
36	100	0.116667	1.3	5.1	36	-5.110430	0.015199
26	70	0.116667	1.3	5.1	26	-5.120925	0.010476
13	40	0.083333	1.3	5.1	13	-5.141338	0.097701
14	40	0.094444	1.3	5.1	14	-5.147760	0.055908
12	40	0.072222	1.3	5.1	12	-5.158752	0.247708
15	40	0.105556	1.3	5.1	15	-5.167356	0.018307
11	40	0.061111	1.3	5.1	11	-5.170102	0.628029
16	40	0.116667	1.3	5.1	16	-5.205845	0.007221
37	100	0.127778	1.3	5.1	37	-5.211725	0.010476
27	70	0.127778	1.3	5.1	27	-5.217093	0.008697
17	40	0.127778	1.3	5.1	17	-5.247752	0.002848
38	100	0.138889	1.3	5.1	38	-5.285479	0.007221
28	70	0.138889	1.3	5.1	28	-5.286623	0.004977
18	40	0.138889	1.3	5.1	18	-5.307340	0.001353
39	100	0.150000	1.3	5.1	39	-5.359563	0.002848
29	70	0.150000	1.3	5.1	29	-5.361907	0.001963
19	40	0.150000	1.3	5.1	19	-5.378538	0.000534
10	40	0.050000	1.3	5.1	10	-5.455930	4.037017
9	10	0.150000	1.3	5.1	9	-9.988070	0.000100
8	10	0.138889	1.3	5.1	8	-14.567807	0.001353
7	10	0.127778	1.3	5.1	7	-20.560388	0.000933
6	10	0.116667	1.3	5.1	6	-26.291141	0.055908
5	10	0.105556	1.3	5.1	5	-31.520993	0.046416
4	10	0.094444	1.3	5.1	4	-36.122634	0.018307
3	10	0.083333	1.3	5.1	3	-41.223198	0.001630
2	10	0.072222	1.3	5.1	2	-48.845707	0.000175
1	10	0.061111	1.3	5.1	1	-56.779745	0.031993
0	10	0.050000	1.3	5.1	0	-64.174907	0.000254

Observation: As can be seen in this table with the best results, the MAE (- best_score) is very low for 14 combinations of the parameters. The best scores are mostly achieved with a high numer of intervals (M), low SD, and quite low penalty term (alpha_parameter).

In [117…

```
def recreate_and_score(model, params, shuffled_index, soft=True):
    # calculate mae on actual test set
    # recreate reps and train/test data
    reps = generate_representaions(**params, soft=soft)

    X_train, X_test, y_train, y_test = split_and_center(
        reps[shuffled_index],
        qm7["T"][0][shuffled_index],
        test_size=0.2,
        shuffle=False,
    )
    y_pred = model.predict(X_test)
    print(f"R2: {r2_score(y_test, y_pred):.3f}")
    print(f"MAE: {mean_absolute_error(y_test, y_pred):.3f}")
    print(f"MSE: {mean_squared_error(y_test, y_pred):.3f}")
    print(f"#coef = 0: {np.sum(lr.coef_ == 0)}")
    print(f"#coef < 1e-10: {np.sum(lr.coef_ < 1e-10)}")
```

The best model in terms of MAE (soft encoding)

We chose the model with the lowest MAE on the validation set and evaluated it on the test data.

In [118…

```
# overall best model in terms of MAE
best_idx = np.argmax([r['search'].best_score_ for r in results])
best_model = results[best_idx]['search']
print("Best model index:", best_idx)
print("Best alpha:", best_model.best_params_['alpha'])
print("Validation MAE:", -best_model.best_score_)
print("Params:", results[best_idx]['params'])
```

```
Best model index: 30
Best alpha: 0.298364724028334
Validation MAE: 4.4924568052875795
Params: {'M': 100, 'STD': 0.05, 'THETA_1': 1.3, 'THETA_M': 5.1}
```

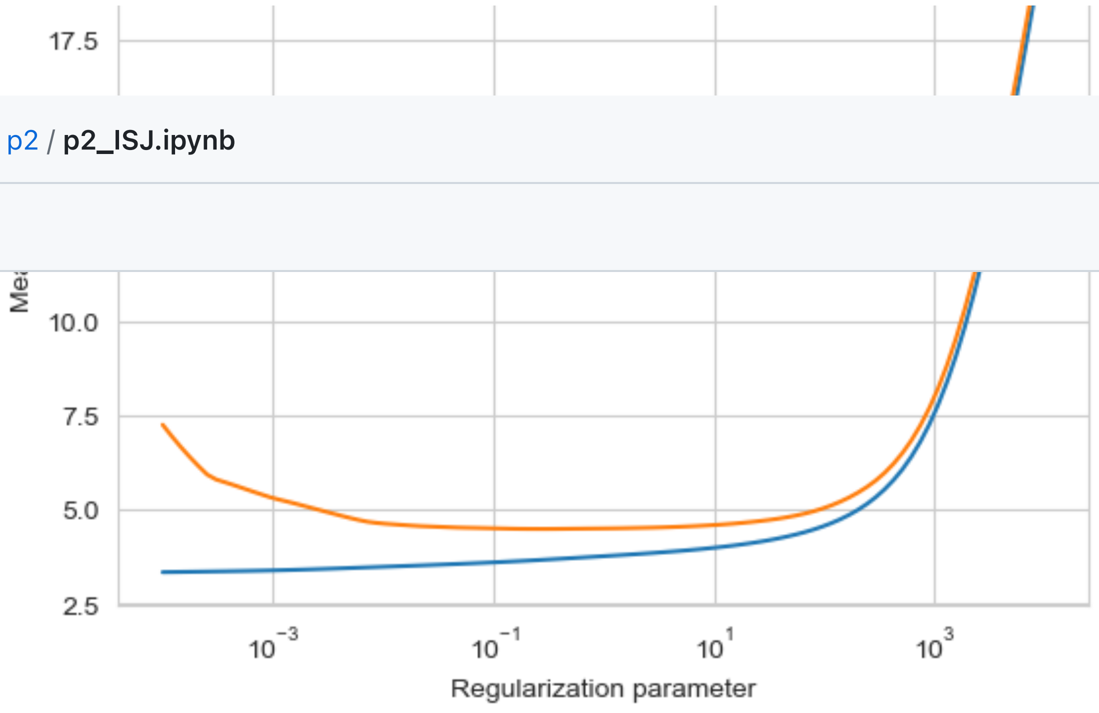
In [119…

```
# Evaluate the model on the test set
recreate_and_score(best_model.best_estimator_, results[best_idx]['params'],
```

```
[1.66982264e-241 1.27870782e-036 5.12640786e-029 ... 0.00000000e+000
 0.00000000e+000 0.00000000e+000]
[  76.50073  -377.50928   191.13074  ...   17.61084   136.21082
  107.940796]
R2: 0.996
MAE: 4.815
MSE: 198.217
#coef = 0: 0
#coef < 1e-10: 5
```

In [120…

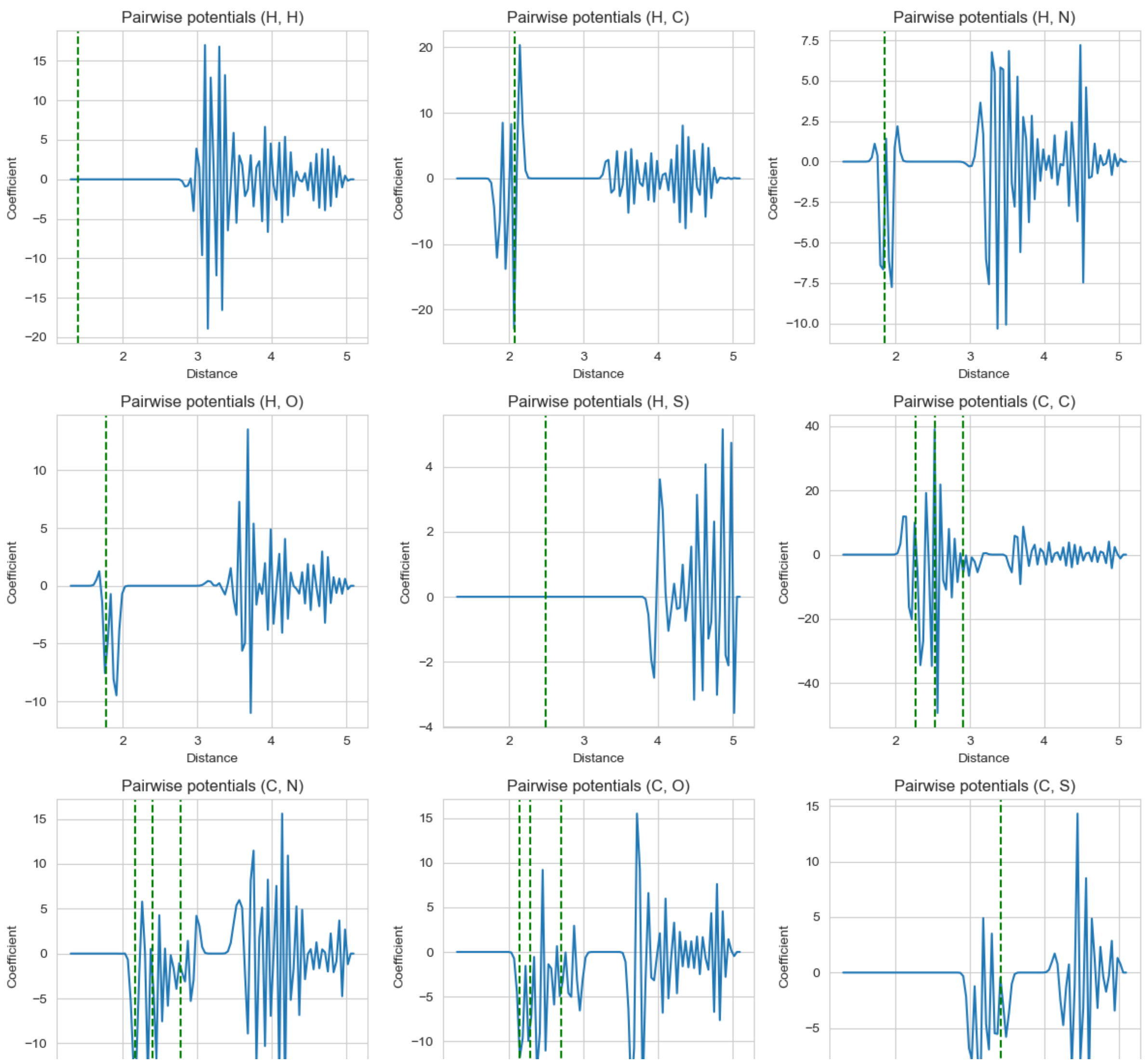
```
# Plot the MAE for different alpha values given the best performing paramet
ax = sns.lineplot(x=alpha_values, y = -best_model.cv_results_['mean_train_s
ax = sns.lineplot(x=alpha_values, y = -best_model.cv_results_['mean_test_sc
ax.set_xscale('log')
ax.set_xlabel('Regularization parameter')
ax.set_ylabel('Mean MAE')
plt.legend(loc='upper left')
plt.show()
```

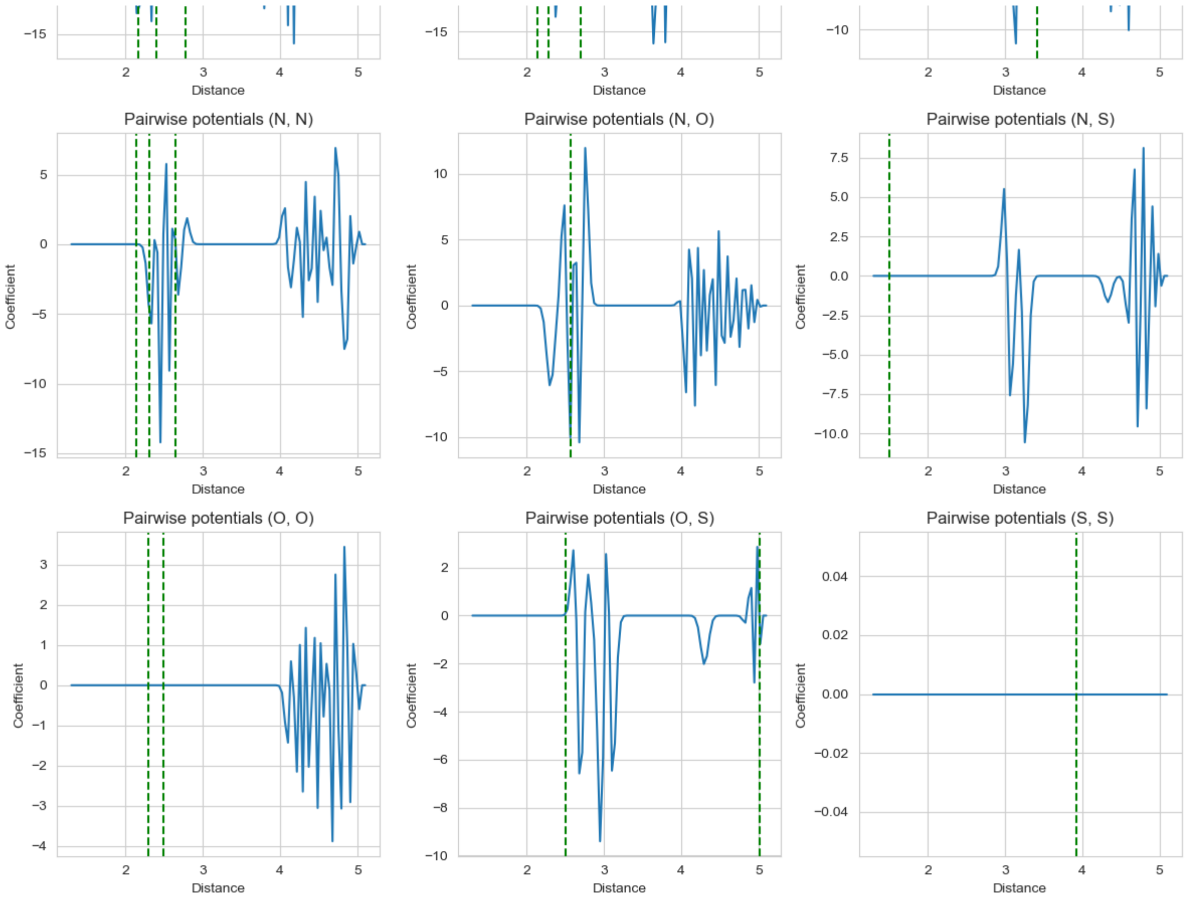


Observation: the optimal regularisation parameter for the validation data set is in the range (0.1, 10).

In [121…

```
# Plot the coefficients
plot_pp(best_model.best_estimator_, **results[best_idx]['params'])
```





Observation: The resulting coefficients' plots suggest that even though our Ridge regression model is highly linear, it is strongly unregularised. This can be observed in the cyclic behaviour of the coefficients' plots.

In addition to that, due to the low SD, the unnatural discontinuities get introduced in the model and the effect of using a Gaussian function is practically invisible.

For comparison: Ridge regression with the hard encoding (same parameters)

In [122…

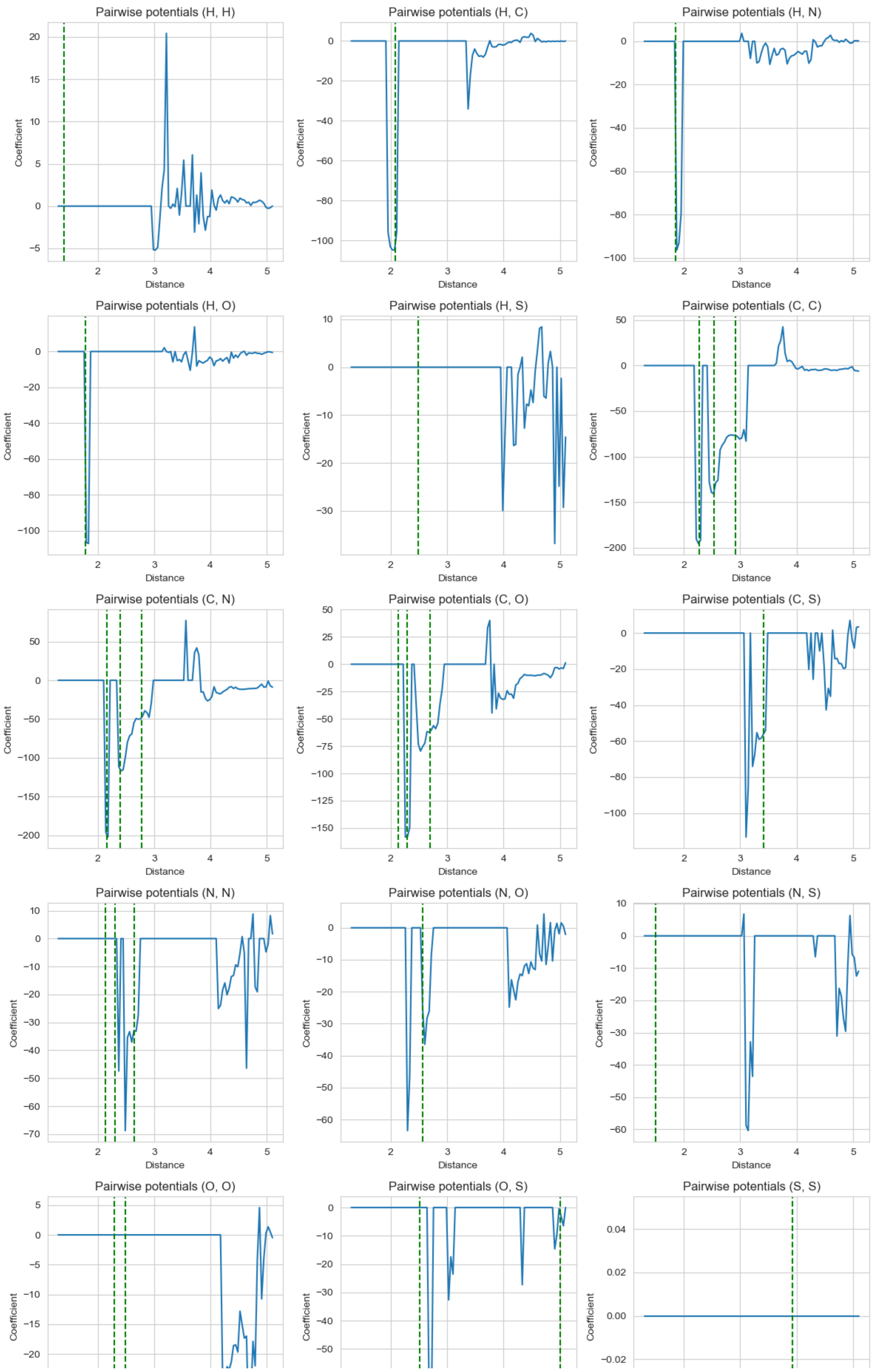
```
def represent_and_split_data(THETA_1, THETA_M, STD, M, soft=True):
    reps = generate_representaions(THETA_1, THETA_M, STD, M, soft=soft)

    X_train, X_test, y_train, y_test = split_and_center(
        reps[shuffled_index],
        qm7["T"][0][shuffled_index],
        test_size=0.2,
        shuffle=False,
    )
    return X_train, X_test, y_train, y_test
```

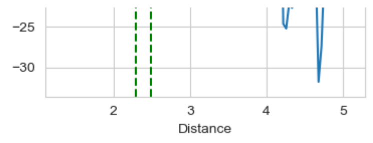
In [123…

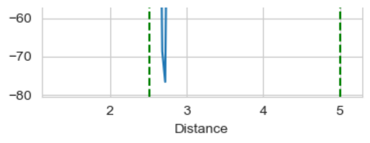
```
THETA_1 = 1.3
THETA_M = 5.1
M = 100
STD = 0.05
ALPHA = 0.298364724028334

X_train, X_test, y_train, y_test = represent_and_split_data(THETA_1, THETA_I
fit_and_eval_model(Ridge, ALPHA, X_train, y_train, X_test, y_test,  M, THET
```

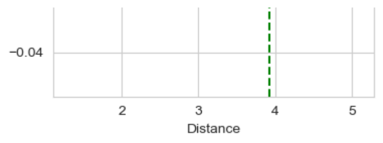
```
[0.          0.          0.          ... 0.00314027 0.          0.          ]
```

```
[0.          0.          0.          ... 0.00031027 0.          0.          ]
[  76.50073  -377.50928   191.13074  ...   17.61084   136.21082
   107.940796]
R2: 0.994
MAE: 5.650
MSE: 306.765
#coef = 0: 957
#coef < 1e-10: 1392
```









Observation: the model performs similarly to the soft Ridge regression. That is because the previous model had a low regularisation term and low variance of the Gaussian function, making it effectively similar to the hard Ridge model seen here.

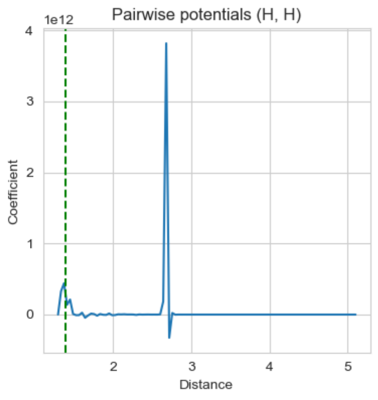
For comparison: Multiple Linear Regression (soft encoding, same parameters)

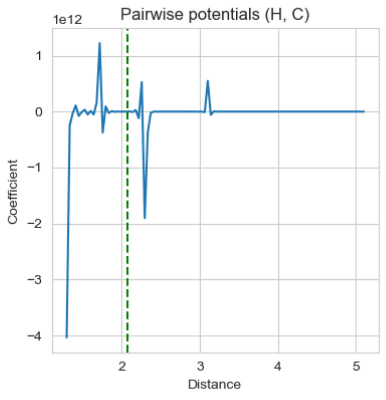
For comparison reasons, we also computed a multiple linear regression which fits the data very poorly indicating that the regularisation in the Ridge regression, even if not very high, very strongly improved the model's predictions.

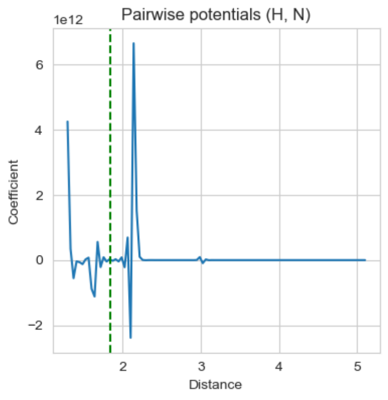
In [124…

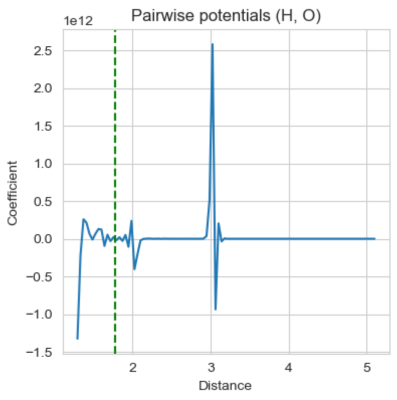
```
X_train, X_test, y_train, y_test = represent_and_split_data(THETA_1, THETA_

lr = LinearRegression()
lr.fit(X_train, y_train)

y_pred_lr = lr.predict(X_test)
print(f"R2: {r2_score(y_test, y_pred_lr):.3f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred_lr):.3f}")
print(f"MSE: {mean_squared_error(y_test, y_pred_lr):.3f}")

plot_pp(lr, M, THETA_1, THETA_M)
```

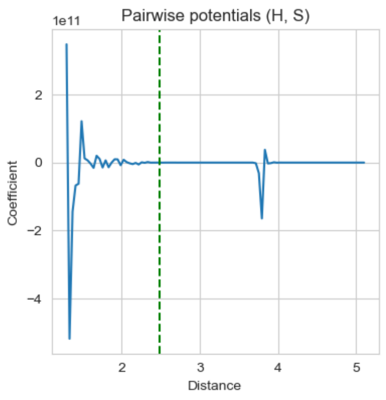
```
[1.66982264e-241 1.27870782e-036 5.12640786e-029 ... 0.00000000e+000
 0.00000000e+000 0.00000000e+000]
[   76.50073   -377.50928    191.13074  ...    17.61084    136.21082
   107.940796]
R2: -1148628053923976448.000
MAE: 6360734709.480
MSE: 5648196928978191751577.000
```

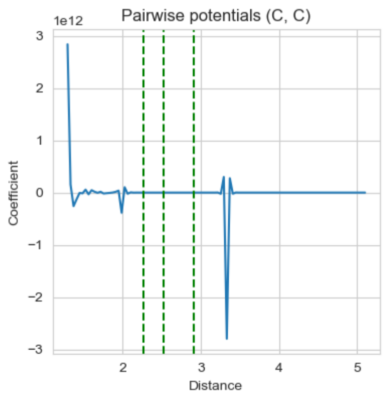


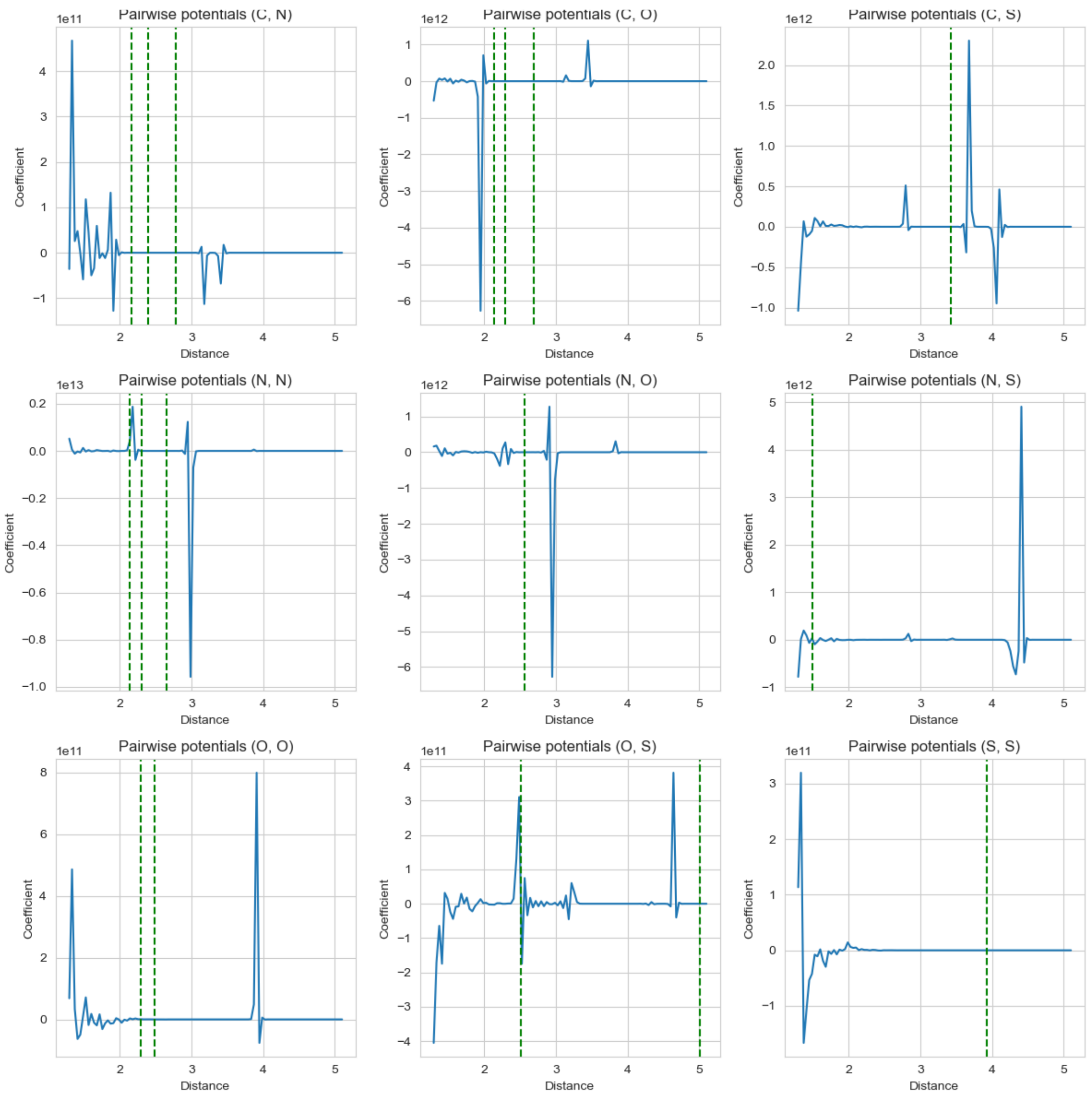












Observation: even though the selected penalty term in the Ridge regression is low, the Multiple linear regression is not able to capture the relationships in the data, and performs much worse in comparison.

3.2 Explanations: Pairs-of-atoms-based Data Representation

Manual parameter search

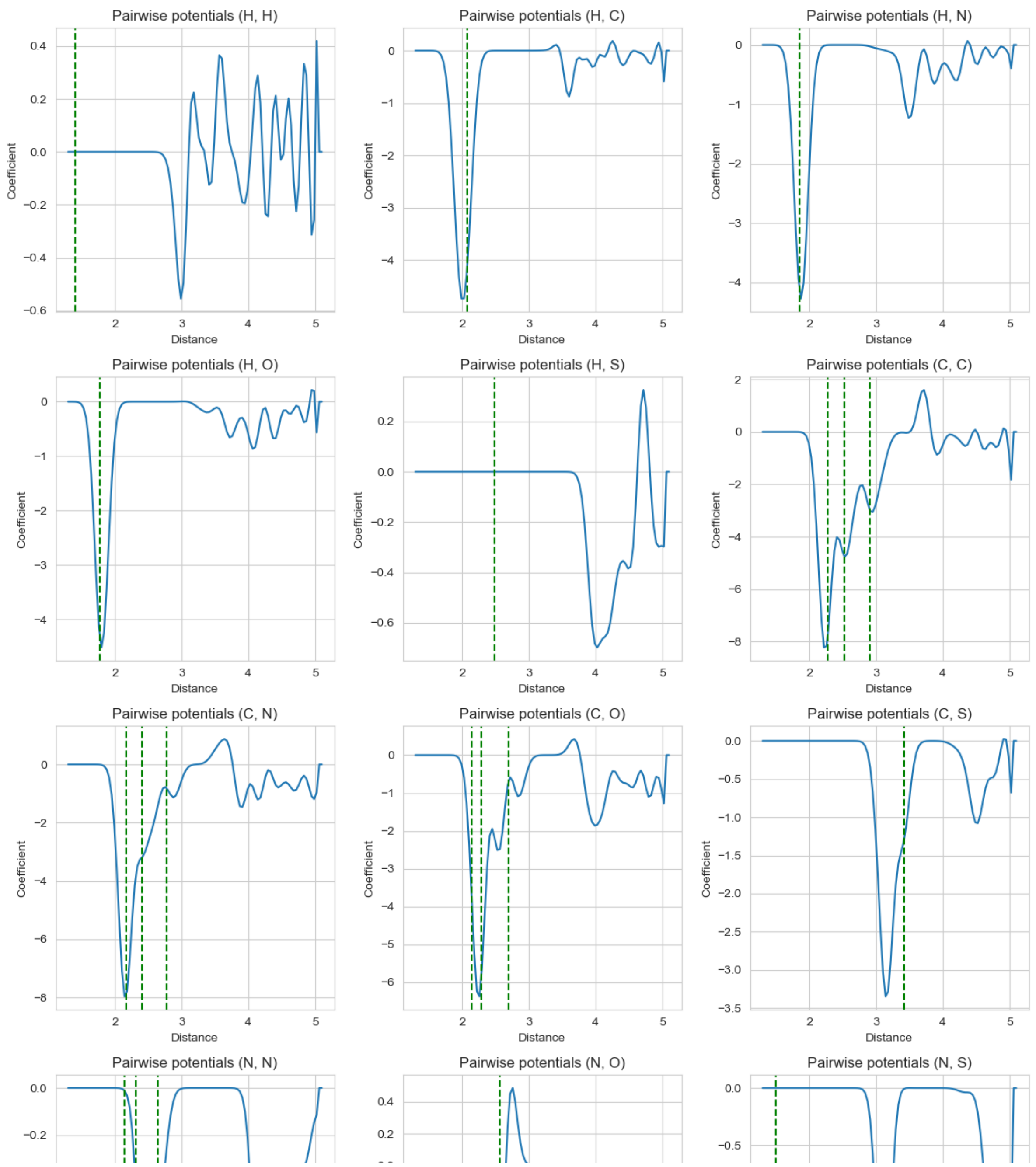
The best model in terms of explainability and external validity (soft encoding)

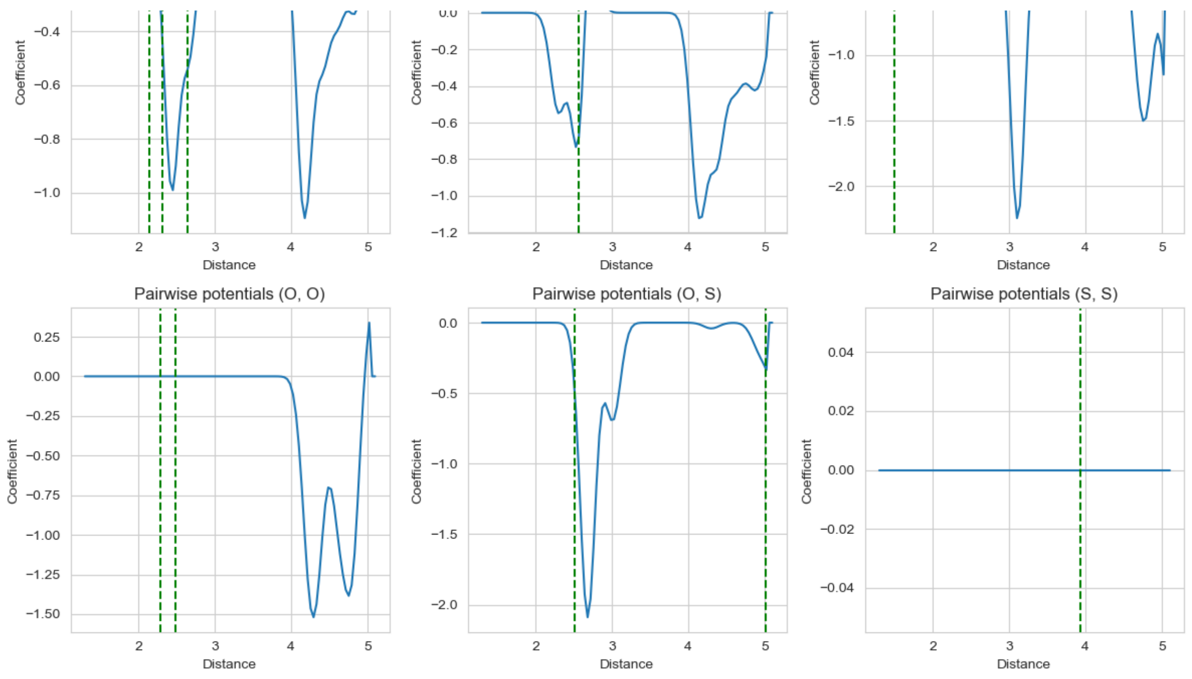
In order to achieve a better model, we first of all increased the SD to 0.1. This way the representation of the data is more smooth. In addition to that, we increased the penalty term to 1000, to achieve a higher regularisation. Both these changes should achieve a model with better explainability of the coefficients - due to smoothness and discarded cyclical behaviour. Also, this way the external validity should be increased too, as the training data-specific behaviour is less pronounced.

In [125…

```
# Approach: look at the models where the # of coefficients higher than 1e-4
#res_df.sort_values(by='n_coef_smaller_1e-4', ascending=False).head(20)

THETA_1 = 1.3
THETA_M = 5.1
M = 100
STD = .1

ALPHA = 1000

X_train, X_test, y_train, y_test = represent_and_split_data(THETA_1, THETA_
fit_and_eval_model(Ridge, ALPHA, X_train, y_train, X_test, y_test,  M, THET
```

```
[2.73525675e-63 1.98552161e-10 1.15771224e-07 ... 0.00000000e+00
 0.00000000e+00 0.00000000e+00]
[  76.50073  -377.50928   191.13074  ...   17.61084   136.21082
  107.940796]
R2: 0.996
MAE: 9.095
MSE: 207.298
#coef = 0: 128
#coef < 1e-10: 1352
```





Observation: The model achieves a somewhat higher MSE than the best performing model, but both the MSE and the explained variance are improved compared to the linear regression with a simple atom-based represenation. In terms of our "sanity check", we can see that some coefficients represent the expected bond lengths very well (e.g. CC), while some of them do not (e.g. HH). Unfortunately, we can not explain the latter. In the consultation we have learned that Ridge regression has an implicit non-local strategy: due to the penalty term it looks at distances that are farther away, but potentially overfits. It might be that some of the variation in the data is specific to our sample and not so representative of the population of the possible molecules.

```
In [125…
```