# MATH 498: Foundations of Machine Learning Winter 2022
# Project 1 Regression

Cooper Stevens (coopstev@umich.edu)
Yuxiang Ying (yingyx@umich.edu)
Sizhuang He (sizhuang@umich.edu)

March 16, 2022

## Experimental Protocol

We aim to predict the heating load of a house given a number of features via linear regression. To begin, we investigate the connections between each feature and the label in physics. We also generate a graph to visualize the relationships with the help of class examples which help us discard some features. Then we add some useful features by operating on different features according to their physical meaning and abandon some duplicated or redundant features. Finally, we train with a number of regression models and compare the evaluation results of each via cross validation to select the final model. The following approaches we describe yield approximately 0.47413 error on the testing set.

## 1 Tools

Python = 3.6.9
numpy = 1.21.5
pyplot from matplotlib = 3.2.2
pandas = 1.3.15
seaborn = 0.11.2
sklearn = 1.0.2
math (version intuited from python version)

## 2 Algorithm

- We used Linear regression, Ridge regression, Ridge regression with polynomial kernel and Gaussian kernel.
- Linear regression is linear model using ordinary least squares with default parameters.
- Ridge regression imposes a penalty on the size of the coefficients with L2 regularization.

- Ridge regression with polynomial kernel and Gaussian kernel map the data into higher dimension and do the linear regression with the help of known inner product. Polynomial kernel is defined as

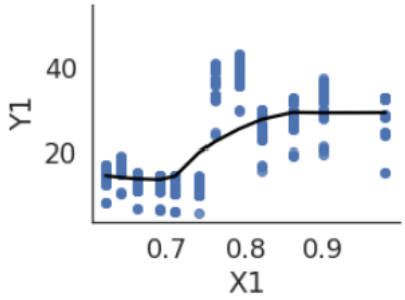
$$K(x, y) = (x^T y + c)^d$$

and $d$ is set as 3 as default, which means the it is mapped to a space with dimension $\binom{N+d}{d} = 2300$, where our feature number $N$ is 22 and degree $d$ is 3. The hyperparameter 'degree' is chosen as 3 since it has better performance than degree 2 and degree 4 has more complexity which leads to overfitting. By Bias-Complexity trade-off, we chose degree as 3. For Gaussian Kernel, we tried the RBF with default length scale 1.0 and find the result not very satisfactory.

## 3 Features

We tried a number of feature transformations and combinations. I will briefly outline each one we tried:

- We split Relative Compactness (X1) into two separate columns (one for X1 $< 0.75$ and another for X1 $\geq 0.75$) because the distribution of the data indicates that there are two separate groups that X1 is divided into w.r.t. Heating Load (Y1) : generally, low X1 corresponds to to low Y1 and high X1 corresponds to high Y1. Entries that are not in the range of the specified column are set to zero, but otherwise the entries are not modified. Displayed below is the split that X1 vs. Y1 experiences.




- We calculate a new column Volume (V) which is calculated by Surface Area / Relative Compactness (X2/X1).

- We calculate another new column Window Surface Area (G) calculated by proportion Glazing Area $\times$ Surface Area (X7$\times$X2).

- We split Glazing Area Distribution (X8) into 6 different buckets : one bucket each for X8 $= 0$, 1, 2, 3, 4, 5. This is because Glazing Area Distribution has only discrete classes that have no relation to the numbers assigned to each.

- We also split Orientation (X6) into 4 different buckets : one bucket each for X6 $= 2$, 3, 4, 5. This is because Orientation has only discrete classes that have no relation to the numbers assigned to each.

- We transform Surface Area (X2) by square rooting all entries. We chose this feature transformation because the rate of change of temperature of a body is proportional to the inverse of the radius

of a body.[1] That is, $\frac{dT}{dt} \propto \frac{1}{r}$. This is a result from Newton's Law of Cooling. And since Surface Area $SA \propto r^2$, we can rearrange to get that $\frac{dT}{dt} \propto \frac{1}{\sqrt{SA}}$. Furthermore, since Heating Load (Y1) is proportional to the rate of heat loss ($\frac{dT}{dt}$), we have that Heating Load has a linear relationship with $\frac{1}{\sqrt{SA}}$. For reasons unexplained, the resultant models actually perform (very slightly) better (approximately 0.01 less error) when we do not invert the result of the square root, so this is the transformation that we have used in our final model.

- Similarly to the transformation for Surface Area, we also transform Volume by cube rooting all entries. This follows a similar reasoning[2]: $\frac{dT}{dt} \propto \frac{1}{r}$ and $V \propto r^3$ so rearrange to get $\frac{dT}{dt} \propto \frac{1}{\sqrt[3]{V}}$. Therefore, Heating Load is proportional to the inverse of the cube root of volume. Again, for reasons unexplained, the resultant models actually perform better (approximately 0.1 less error) when we do not invert the result of the square root, so this is the transformation that we have used in our final model.

- Lastly, we drop a number of features:

  - We drop proportion Glazing Area (X7) because the actual glazing area (G) is much more useful when calculating heat loss.
  - We drop Orientation (X6) and Glazing Area Distribution (X8) because we have bucketed them into other columns.
  - We drop Overall Height (X5) because we do not know how to use this parameter nor how it was measured (some of the heights given are negative!).
  - Lastly, we drop Relative Roof Area (X9) because we already have accounted for Roof Area (X4) by including X4, so this feature seems redundant.

# 4 Parameters

We compared the performance of 4 classes of models, Linear Regression, Ridge Regression, Kernel Ridge Regression with polynomial kernel, and Lasso Regression, on our data during the cross validation process. The following is a step-by-step description of the cross validation process we applied.

1. Split the data into 5 folds. Each time we run the training algorithm, 1 of the folds will be the test fold and the rest are the training folds.

2. For a given class of models from our four candidates, train on the training folds and test on the test fold and record the test error. The loss function that we use is the Root Mean Squared Error (RMSE).

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

3. Repeat Step 2 for another 4 times on the same class of models

[1] Rennie, J. (2014, February 22). *Surface Area: Volume and its relation with heat*. Physics Stack Exchange. Retrieved March 15, 2022.

[2] Ibid.

4. Calculate the cross validation error of this class of models. The cross validation error is obtained by averaging the test errors that we calculated previously.

5. Repeat Step 2, 3 and 4 for another 3 times, one for each class of models.

6. Compare the cross validation error of these 4 classes of models and choose the one with the least cross validation error to be our final model and train a model from it.

After going through the cross validation process as described above, we choose the Kernel Ridge Regression model to be our final model.During the cross validation process, we have trained and evaluated a number of models. The training process, which is essentially searching for parameters of the model, is integrated in functions of the $sklearn$ library.

After dropping certain features and creating certain new features, we have 16 features in total. We can view the features as a 16-dimensional vector for each data point. For Linear Regression, we want to obtain an affine function of the form $f(x) = \omega^T x + b$. The parameters to search for are two 16-dimensional vector $\omega$ and $b$. For Ridge Regression and Lasso Regression, we are adding a regularization function to the minimization problem but the parameters to search for do not change. This the same for the Kernel Ridge Regression, in which case we are using kernels as a way to compare features.

# 5 Lessons Learned

We tried removing additional features such as Wall Area (X3) and Roof Area (X4) because these almost exactly determine Surface Area (X2), so we figured that including all three features would be redundant. It turns out, however, than including all of them (and transforming Surface Area) encodes valuable information for predicting Heating Load. After testing all combinations of including/exclusion with these features, we found it best to include all three features in our final model. Perhaps the small nuances in the differences from the dependency that these features normally experience are very telling of the heating load of the house.

Before we bucketed Orientation (X5) and Glazing Area Distribution (X8), we tried removing them entirely because the orientation of the house seems useless when we don't know what the orientation is measures with respect to. And the classification of Glazing Area Distribution also seems useless without know what each class means. Nonetheless, our models performs much better after bucketing both of these features into columns for each of their possible values. Perhaps these classifications encode something essential about the heating load, such as how the wind hits the heat-loss-vulnerable parts of the house.

We also considered interpreting Overall Height (X5) as an altitude from sea level. This would enable us to calculate barometric pressure for each house, and therefore the number of atom collisions with the house per time unit (this would also depend on the surface area/glazing area). This seems like it would be an important factor because atom collisions is how the house loses heat (heat loss is proportional to the number of atom collisions). And so we expected the number of atom collisions to be a very informative feature. But once implemented, the test scores showed that was, indeed, not a helpful feature. This probably failed because this is is the incorrect interpretation of Overall Height.

# Project 1 Code Base

March 16, 2022

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import sklearn as sk
import math
```

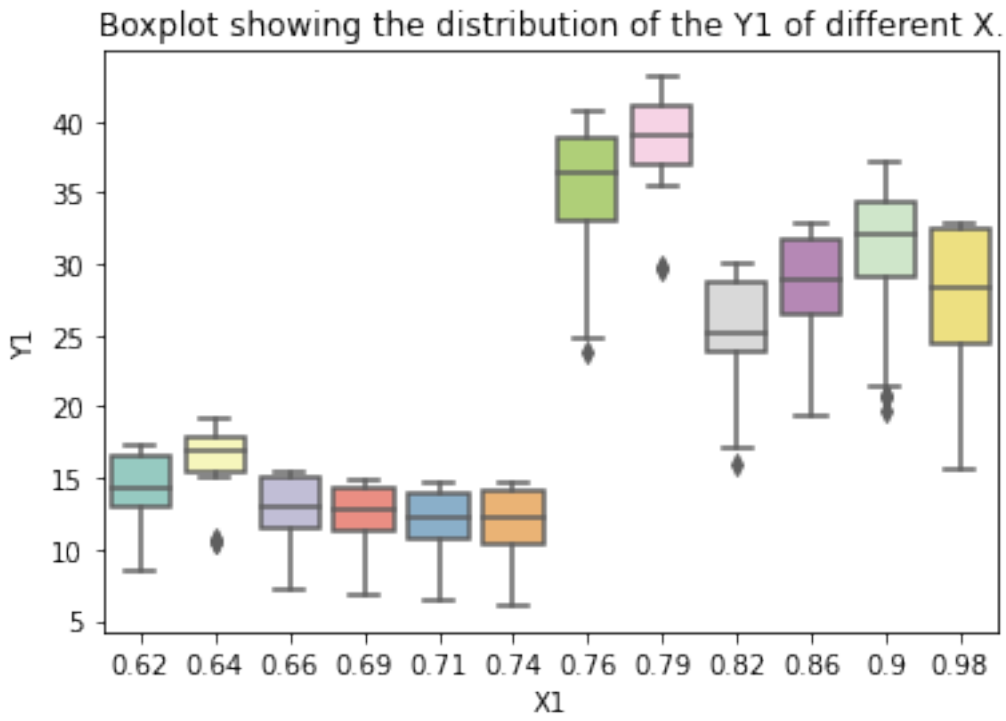
[2]:
```
df = pd.read_csv(r"2309_train.csv", index_col=0)
```

[3]:
```
df['X8'].unique()
```

[3]:
```
array([2, 5, 3, 0, 4, 1])
```

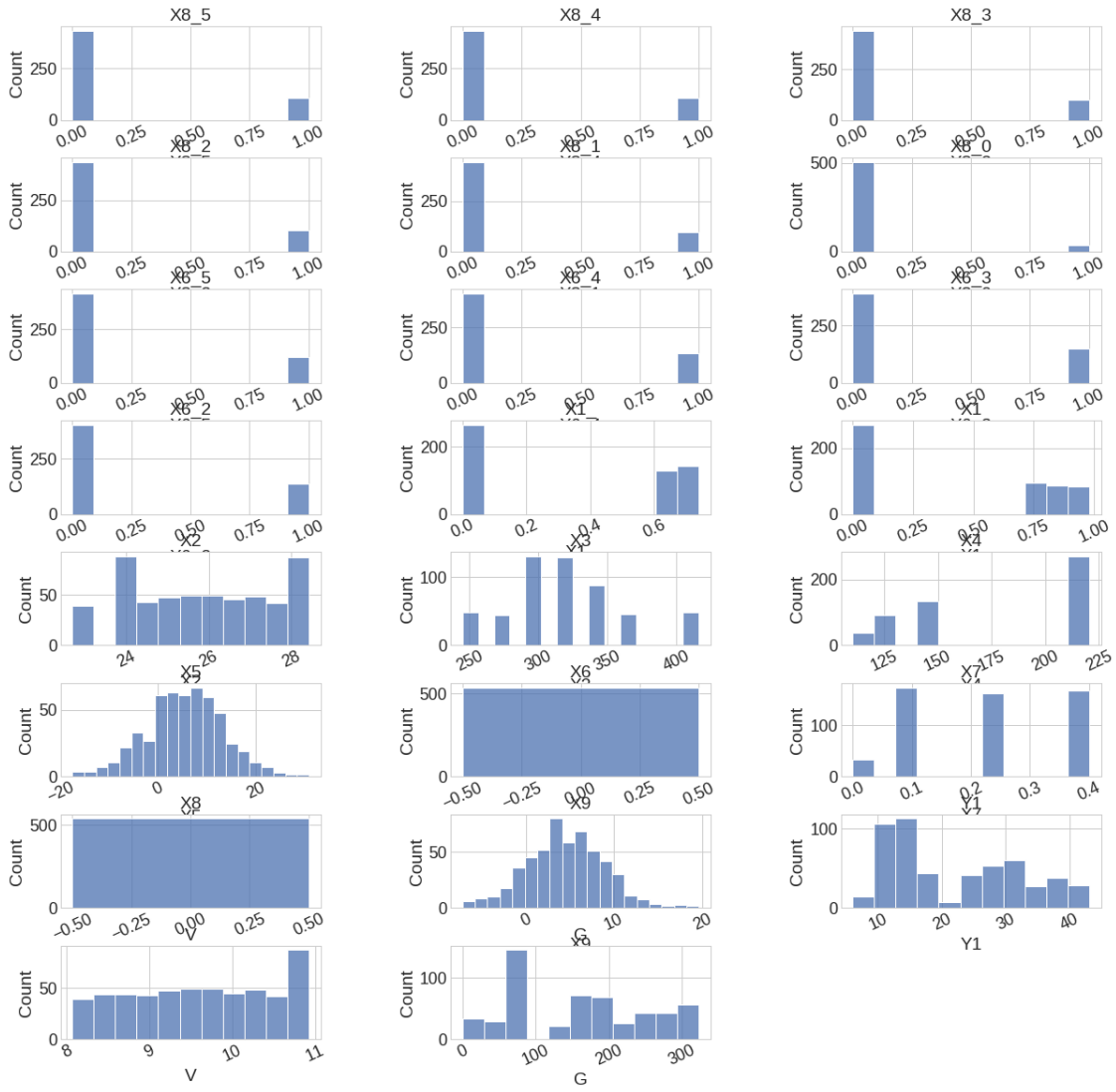
[4]:
```
# sns.scatterplot(data = df, x = "X9", y = "Y1")
```

[5]:
```
sns.boxplot(x="X1", y="Y1", data=df ,palette="Set3");
plt.title("Boxplot showing the distribution of the Y1 of different X.");
```




[27]:
```
def plot_distribution(dataset, cols=5, width=20, height=15, hspace=0.2,␣
↪wspace=0.5):
    plt.style.use('seaborn-whitegrid')
    fig = plt.figure(figsize=(width,height))
    fig.subplots_adjust(left=None, bottom=None, right=None, top=None,␣
↪wspace=wspace, hspace=hspace)
    rows = math.ceil(float(dataset.shape[1]) / cols)
    for i, column in enumerate(dataset.columns):
        ax = fig.add_subplot(rows, cols, i + 1)
        ax.set_title(column)
        if dataset.dtypes[column] == object:
            g = sns.countplot(y=column, data=dataset)
            substrings = [s.get_text()[:18] for s in g.get_yticklabels()]
            g.set(yticklabels=substrings)
            plt.xticks(rotation=25)
        else:
            g = sns.histplot(dataset[column])
            plt.xticks(rotation=25)

plot_distribution(df, cols=3, width=20, height=20, hspace=0.4, wspace=0.5)
```

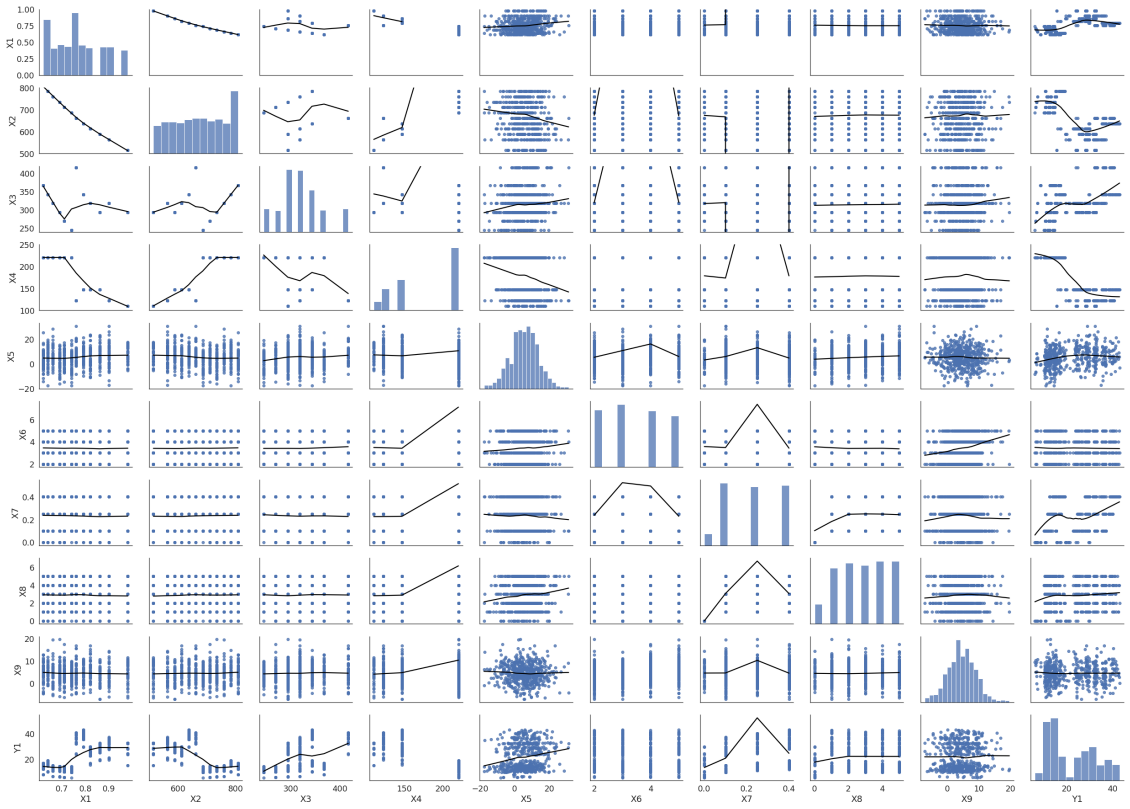



[7]:
```
sns.set(style='white', font_scale=1.6)
g = sns.PairGrid(df, aspect=1.4, diag_sharey=False)
g.axes[0,0].set_ylim((0,1))
g.axes[1,0].set_ylim((500,800))
g.axes[2,0].set_ylim((240,420))
g.axes[3,0].set_ylim((100,250))

g.map_lower(sns.regplot, lowess=True, ci=False, line_kws={'color': 'black'})
g.map_diag(sns.histplot, kde_kws={'color': 'black'})
g.map_upper(sns.regplot, lowess=True, ci=False, line_kws={'color': 'black'})
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
```

```
public API at pandas.testing instead.
  import pandas.util.testing as tm
```




[8]:
```
df_initial = df.copy()
```

# 1 Reset df

[9]:
```
df = df_initial.copy() # initialize
```

# 2 Intervals

[10]:
```
def intervalize(datain, col, val):
  datain.insert(0, col + '_', 0)
  for i, j in datain.iterrows():
    if (datain.loc[i, col] < val):
      datain.loc[i, col + '_'] = datain.loc[i, col] # the new row will have the␣
 ↪value if it is in some range
      datain.loc[i, col] = 0  # the original row have value 0 replaced
def bucket(datain, col, val):
  datain.insert(0, col + "_" + str(val), 0)
```

```
  for i, j in datain.iterrows():
    if (datain.loc[i, col] == val):
      datain.loc[i, col + "_" + str(val)] = 1
      datain.loc[i, col] = 0
```

[11]:
```
df['V'] = df['X2']/df['X1'] # BEST
df['G'] = round(df['X7']*df['X2'], 2) # BEST
# df['P'] = (((1 - 0.0065*df['X5']/288.15).map(lambda x: x ** (5.25588))) *␣
↪df['X2']).map(lambda x: math.sqrt(x)) # not work
intervalize(df, 'X1', 0.75)
# intervalize(df, 'X7', 0.25) # not work
for i in range(2, 6): # better kaggle result but poorer local CV error
  bucket(df, 'X6', i)
for i in range(6):
  bucket(df, 'X8', i)
# df['X4'] = df['X4'].map(lambda x: math.sqrt(x)) # not work
# df['X3'] = df['X3'].map(lambda x: math.sqrt(x)) # not work
df['X2'] = df['X2'].map(lambda x: math.sqrt(x)) # BEST
df['V'] = df['V'].map(lambda x: x ** (1/3)) # BEST
# intervalize(df, 'V', 9.75) # not work
# df['X7_'] = df['X7_'].map(lambda x: x ** 2) # not work
# df['1/V'] = df['V'].map(lambda x: 1/x)  # not work
# df['V3'] = df['V'].map(lambda x: x**3)  # not work
# df['1/X2'] = df['X2'].map(lambda x: 1/x) # not work
```

## 3 Data_process (discard X2, X5, X9)

[12]:
```
def process_data_frame(data_in):
    dataframe = data_in.copy()

    # attributes_to_drop = ["X5", "X9", "X3", "X4"]
    # attributes_to_drop = ["X2", "X9", "X5", "X6", "X8"]
    # attributes_to_drop = ["X3", "X4", "X9", "X5", "X6", "X8"]
    # attributes_to_drop = ["X9", "X5", "X6", "X8"]
    attributes_to_drop = ["X9", "X5", "X6", "X8", "X7"]
    # attributes_to_drop = ["X9", "X5", "X6", "X8", "X7", "X2"]
    dataframe.drop( attributes_to_drop,axis=1,inplace=True)
    return dataframe

def get_features_and_target(dataframe,target="Y1"):
    label = dataframe[target].copy()
    features = dataframe.drop(target,axis=1).copy()
    return features, label
```

[13]:
```
df_processed = process_data_frame(df)
target = "Y1"
X, y = get_features_and_target(df_processed, target)
```

## 4 Learn (Cross Validation)

[14]:
```
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.kernel_ridge import KernelRidge
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
from collections import defaultdict
from sklearn.gaussian_process.kernels import RBF

kf = KFold(n_splits=5)

train_err = defaultdict(list)
crossval_err = defaultdict(list)
K = 1.0 * RBF(1.0)

candidate_models= {'Model 1': LinearRegression(), 'Model 2':Ridge(),\
                   'Model 3': KernelRidge(kernel='poly', degree=3)}


for model_name, candidate_model in candidate_models.items():
    print("Model type:", candidate_model)
    for train_index, test_index in kf.split(X):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        model = candidate_model.fit(X_train, y_train)

        predictions_training = model.predict(X_train)
        predictions = model.predict(X_test)

        rmse_training = np.
 ↪sqrt(mean_squared_error(predictions_training,y_train))
        rmse = np.sqrt(mean_squared_error(predictions,y_test))
        print(rmse_training, rmse)

        train_err[model_name].append(rmse_training)
        crossval_err[model_name].append(rmse)
```

```
Model type: LinearRegression()
2.3613428039376747 2.6398333617205205
2.4444596373636815 2.3193587238589313
```

```
2.449769494452021 2.3100222209326415
2.3747815446701166 2.58435172767472
2.4017895591974945 2.4779624921704184
Model type: Ridge()
2.869598522610908 3.256306286638506
2.9507952109408393 2.78561400213284
3.030294293832173 2.513076374958124
2.8777148662398666 3.1822569120404967
2.875487573372099 3.1295835591797623
Model type: KernelRidge(kernel='poly')
0.3214408310898987 0.5490080378110873
0.328322543643545 0.5313724852481126
0.3127678337428317 0.6358097241589209
0.3309994275863527 0.5301122268906794
0.325774052375748 0.5272269564319272
```

## 5 Print

[15]:
```
for key in crossval_err.keys():
    print(f'Training error of {key}: {np.mean(train_err[key])}')
    print(f'Cross-val error of {key}: {np.mean(crossval_err[key])}')
```

```
Training error of Model 1: 2.4064286079241977
Cross-val error of Model 1: 2.4663057052714463
Training error of Model 2: 2.920778093399177
Cross-val error of Model 2: 2.9733674269899457
Training error of Model 3: 0.3238609376876752
Cross-val error of Model 3: 0.5547058861081455
```

## 6 Test Init (reset test data)

[16]:
```
test_set = pd.read_csv(r"2309_test_no_labels.csv", index_col=0)
```

[17]:
```
test_init = test_set.copy()
```

[18]:
```
test_set = test_init.copy()
```

## 7 Para

[19]:
```
test_set['V'] = test_set['X2']/test_set['X1'] # BEST
test_set['G'] = round(test_set['X7']*test_set['X2'], 2) # BEST
intervalize(test_set, 'X1', 0.75)
for i in range(2, 6):
  bucket(test_set, 'X6', i)
for i in range(6):
```

```
  bucket(test_set, 'X8', i)
test_set['X2'] = test_set['X2'].map(lambda x: math.sqrt(x)) # BEST
test_set['V'] = test_set['V'].map(lambda x: x ** (1/3)) # BEST
```

[20]:
```
test_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 231 entries, 597 to 102
Data columns (total 22 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   X8_5    231 non-null    int64  
 1   X8_4    231 non-null    int64  
 2   X8_3    231 non-null    int64  
 3   X8_2    231 non-null    int64  
 4   X8_1    231 non-null    int64  
 5   X8_0    231 non-null    int64  
 6   X6_5    231 non-null    int64  
 7   X6_4    231 non-null    int64  
 8   X6_3    231 non-null    int64  
 9   X6_2    231 non-null    int64  
 10  X1_     231 non-null    float64
 11  X1      231 non-null    float64
 12  X2      231 non-null    float64
 13  X3      231 non-null    float64
 14  X4      231 non-null    float64
 15  X5      231 non-null    float64
 16  X6      231 non-null    int64  
 17  X7      231 non-null    float64
 18  X8      231 non-null    int64  
 19  X9      231 non-null    float64
 20  V       231 non-null    float64
 21  G       231 non-null    float64
dtypes: float64(10), int64(12)
memory usage: 49.6 KB
```

## 8 Test

[21]:
```
processed_test_set = process_data_frame(test_set)
```

[22]:
```
best_model = KernelRidge(kernel='poly').fit(X, y)
```

[23]:
```
prediction_test = best_model.predict(processed_test_set)
```

[24]:
```
processed_test_set['Y1']=prediction_test #adding the column to the test_set
```

[25]:
```
processed_test_set['Y1'].to_csv('predictions_test.csv',index_label='ID') # save
↪CSV that can be dumped
```

## 9 Result

[26]:
```
result = pd.read_csv(r"predictions_test.csv", index_col=0)
result
```

[26]:

| ID | Y1 |
|---|---|
| 597 | 15.102737 |
| 8 | 18.709656 |
| 625 | 23.451622 |
| 36 | 32.552849 |
| 452 | 12.730408 |
| .. | … |
| 71 | 32.769699 |
| 106 | 23.387169 |
| 270 | 41.764061 |
| 435 | 28.292786 |
| 102 | 10.120949 |

[231 rows x 1 columns]