ChBE 4746/6746

Homework 3 Solution

Fani Boukouvala, Pengfei Cheng, Zachary Kilwein

Update date: April 11, 2021

Problem 1 Sampling

- **Sampling strategy**: Latin Hypercube sampling, grid designs, or other strategies sampling through the whole region
- **Sampling number**: no less than 10, and is not unreasonably large
- **Scaling**: recommended (updated: April 11)

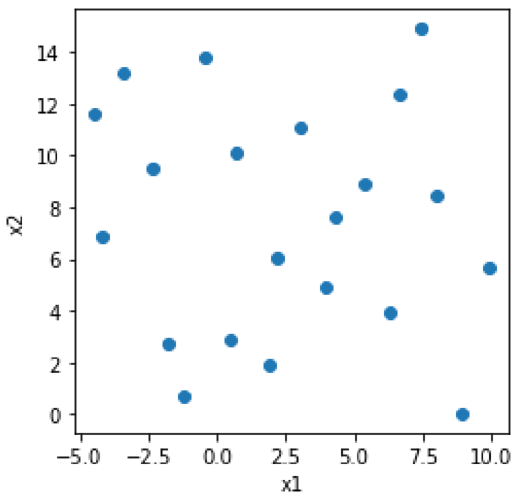
As we collect the data from simulation instead of experiments, we can simply use Latin Hypercube Sampling (LHS) strategy to collect data. Since the input dimension is 2, 20 sampling points are enough. Other sampling strategies are also acceptable, as long as the samples span the whole feasible region. The minimal sampling number is 10. There is no upper bound for the sampling number, but it should be a reasonable number that does not cause the computational cost of following problems prohibitively high.

The accuracy of surrogate models is sensitive to order of magnitudes of data. In Problems 1-4, the inputs are in the same order of magnitude, so it is also acceptable to use unscaled data for model fitting. (updated: April 11) The default generated inputs from `pyDOE.LHS` are directly scaled between 0 and 1, and min-max scaling for the corresponding outputs are enough.

Corresponding code:

```python
import numpy as np
import pyDOE as doe
import matplotlib.pyplot as plt

# Define the black box function
# input: 1D array
def blackbox(x):
    return 10 + 10 * (1 - 1/8/np.pi) * np.cos(x[:,0]) + (x[:,1] - (5.1/16/(np.pi**2)) * (x[:,0])**2
    ↪  + 5 / np.pi * x[:,0] - 6) ** 2

# sampling
# ---------------------------------------------------------------------------------
# LHS
dim = 2
sample = 10 * dim
# this return scaled inputs, dimension: (20, 2)
x_scaled = doe.lhs(dim, samples=sample, criterion='maximin')
# ---------------------------------------------------------------------------------
# # grid designs
# dim = 2
# N = 10
# x_1 = np.linspace(0, 1, N)
# x_2 = np.linspace(0, 1, N)
# xx_1, xx_2 = np.meshgrid(x_1, x_2)
# x_scaled = np.stack((xx_1.flatten(),xx_2.flatten()), axis=-1)
# ----------------------------------------------------------------------------


# unscale input between the actual bounds: -5 <= x_1 <= 10, 0 <= x_2 <= 15
bounds = np.array([[-5, 10], [0, 15]]).T
x = x_scaled.copy()
for i in range(dim):
    x[:, i] = x_scaled[:, i] * (bounds[1, i] - bounds[0, i]) + bounds[0, i]

# call the blackbox function to generate outputs
y = blackbox(x)

# scale y
y_scaled = (y - min(y)) / (max(y) - min(y))

# optional: visualization of sampling points
fig, ax = plt.subplots(figsize=plt.figaspect(1.0))
ax.scatter(x[:, 0], x[:, 1])
ax.set(xlabel='x1', ylabel='x2')
plt.show()
```

Output:



Problem 2 Generalized linear regression (GLR) & LASSO

2(a) Selecting better feature set

- **Results may vary**: average error, better feature set, and the minimal objective value of the surrogate model may be different because of sampling
- **Hyperparameter tuning**: no predetermined range for hyperparameter λ, general grid-search or trial-and-error is enough
- **Accuracy metric**: sum of squared errors of test set; other reasonable metrics are also acceptable
- **Workflow**:
  1. Determine set Λ of potential λ values
  2. Split training-test data for 5-fold CV
  3. For each feature set, repeat:
     - For each λ in Λ, repeat:
       - For each fold, repeat:
         - build and solve generalized linear regression + LASSO model
         - calculate test error
       - calculate average error
     - find the lambda with the lowest average error
  4. find the better feature set with the lower lowest average error

Formulation of generalized linear regression model with LASSO (with $|b_j|$ replaced with b_j^+, b_j^-):

$$
\begin{aligned}
\min\ & \sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\lambda\sum_{j=0}^{m}(b_j^+ + b_j^-)\\
\text{s.t. } & \hat{y}_i=\sum_{j=0}^{m} b_j X_{ij}, i=1,2,\ldots,n\\
& b_j = b_j^+ - b_j^-, i=1,2,\ldots,n\\
& b_j^+, b_j^- \geq 0, i=1,2,\ldots,n,
\end{aligned}
$$

where

- $i \in \{1,\ldots,n\}$: index of sample points
- $j \in \{1,\ldots,m\}$: index of features
- y_i: output of sample point i
- X_{ij}: input feature j of sample point i
- $\hat{y}_i$: predicted value at sample i, variable
- b_j: weight of feature j, variable

To find the better feature set, we can compare the objective value of the corresponding models. Here the objective is directly used as the metric. The tuning of hyperparameter λ is not the focus of this problem, so a grid search or a trial-and-error search for best λ value is enough. Ideally, to identify the best (feature set, λ) pair, there should be 3 loops: one over two feature sets, one over λ values, and one over 5 folds for 5-fold CV. The code structure does not have to follow the exactly same order, but should be identical to this.

The final results may vary because of the difference of sampling in Problem 1, So either feature set may have a better objective value with a certain λ value.

Corresponding code:

```python
import pyomo.environ as pe

# define two functions to generate new features
def mat_1(x):
    """Return matrix of 1, x1, x2, x1^2, x2^2, x1^4, cos(x1), x1 * x2. Dimension: (n_samples, 8)"""
    m, n = np.shape(x)
    mat = np.array([np.ones(m), x[:,0], x[:,1], x[:,0]**2, x[:,1]**2, x[:,0]**4, np.cos(x[:,0]),
    ↪  x[:,0]*x[:,1]]).T
    return mat

def mat_2(x):
    """Return matrix of 1, x1, x2, x1^2, x1^3, x2^2, x2^3, x1 * x2, x1^2 * x2, x1 * x2^2. Dimension:
    ↪  (n_samples, 10)"""
    m, n = np.shape(x)
    mat = np.array([np.ones(m), x[:,0], x[:,1], x[:,0]**2, x[:,0]**3, x[:,1]**2, x[:,1]**3,
    ↪  x[:,0]*x[:,1],(x[:,0]**2) * x[:,1], x[:,0] * (x[:,1]**2)]).T
    return mat

# generate new features
X_new = {}
X_new['I1'] = mat_1(x_scaled)
X_new['I2'] = mat_2(x_scaled)

# --------------------------------------------------------------------------------

# 2(a)

# CV from sklearn
from sklearn.model_selection import KFold

# create k-fold object
kf = KFold(n_splits=5, shuffle=True)

# lambda value set
lbda_set = np.logspace(-4, 0, num=11)

# create dict for average error of each feature set, and each lambda value
overall_avg_error = {}

# train the model for both feature sets
for feature_i, X in X_new.items():

    print(f'Feature set {feature_i}:')

    # create index list for features
    feature_list = list(range(X.shape[1]))

    # create dict for average error of each lambda value
```

```
avg_error = {}

# grid-search lambda
for lbda in lbda_set:


    # create dictionary to record trained model for each fold
    CV_models = []

    # 5-fold CV
    for train_index, test_index in kf.split(X):

        # split training and test set
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y_scaled[train_index], y_scaled[test_index]

        # create index list for training points
        train_point_list = list(range(X_train.shape[0]))

        # ----------------------------------------------------------------------------------
        # Pyomo model
        m = pe.ConcreteModel()

        # define variables
        m.b = pe.Var(feature_list, within=pe.Reals)
        m.y_p = pe.Var(train_point_list, within=pe.Reals)
        m.b_p = pe.Var(feature_list, within=pe.NonNegativeReals)
        m.b_n = pe.Var(feature_list, within=pe.NonNegativeReals)

        # define objective function
        def obj_rule(m):
            return sum((m.y_p[i] - y_train[i]) ** 2 for i in train_point_list) + lbda *
            ↪  sum(m.b_p[j] + m.b_n[j] for j in feature_list)
        m.obj = pe.Objective(rule=obj_rule)

        # define constraints
        def con_rule1(m, i):
            return m.y_p[i] == sum(m.b[j] * X_train[i, j] for j in feature_list)
        m.con1 = pe.Constraint(train_point_list, rule=con_rule1)
        def con_rule2(m, j):
            return m.b[j] == m.b_p[j] - m.b_n[j]
        m.con2 = pe.Constraint(feature_list, rule=con_rule2)

        # solve model using ipopt
        solver = pe.SolverFactory('ipopt')
        solver.solve(m)
        # ----------------------------------------------------------------------------------

        # record model for this fold
        tmp_model = {}
        # record objective value
        tmp_model['obj'] = pe.value(m.obj)
        # record parameter value
        tmp_model['b'] = np.empty((len(feature_list)))
```

```
                for i in feature_list:
                    tmp_model['b'][i] = pe.value(m.b[i])
                # calculate test error
                tmp_model['error'] = sum((np.matmul(X_test, tmp_model['b'].T) - y_test) ** 2)
                # record the model in CV_models
                CV_models.append(tmp_model)

            # calculate average error of 5-fold CV
            avg_error[lbda] = sum(model['error'] for model in CV_models) / len(CV_models)

            print(f'\tlambda = {lbda:.1e}\taverage error = {avg_error[lbda]:.5f}')

        # record error in overall_avg_error
        overall_avg_error[feature_i] = avg_error

    # compare errors
    best_error = {'I1': {}, 'I2': {}}
    best_error['I1']['lambda'] = min(overall_avg_error['I1'], key=overall_avg_error['I1'].get)
    best_error['I1']['error'] = overall_avg_error['I1'][best_error['I1']['lambda']]
    best_error['I2']['lambda'] = min(overall_avg_error['I2'], key=overall_avg_error['I2'].get)
    best_error['I2']['error'] = overall_avg_error['I2'][best_error['I2']['lambda']]

    if best_error['I1']['error'] <= best_error['I2']['error']:
        better_feature_i = 'I1'
    else:
        better_feature_i = 'I2'

    # record better feature set and corresponding lambda
    better_X = X_new[better_feature_i]
    best_lambda = best_error[better_feature_i]['lambda']

    print(f"\nBest feature set: {better_feature_i}, with lambda =
    ↪  {best_error[better_feature_i]['lambda']:.1e}, "
          f"error = {best_error[better_feature_i]['error']:.5f}.")
```

Output:

```
    Feature set I1:
            lambda = 1.0e-04        average error = 0.00523
            lambda = 2.5e-04        average error = 0.00236
            lambda = 6.3e-04        average error = 0.00185
            lambda = 1.6e-03        average error = 0.00247
            lambda = 4.0e-03        average error = 0.01293
            lambda = 1.0e-02        average error = 0.02773
            lambda = 2.5e-02        average error = 0.03000
            lambda = 6.3e-02        average error = 0.04722
            lambda = 1.6e-01        average error = 0.08754
            lambda = 4.0e-01        average error = 0.12423
            lambda = 1.0e+00        average error = 0.17874
    Feature set I2:
            lambda = 1.0e-04        average error = 0.00588
            lambda = 2.5e-04        average error = 0.00828
```

```
        lambda = 6.3e-04        average error = 0.00380
        lambda = 1.6e-03        average error = 0.01038
        lambda = 4.0e-03        average error = 0.02099
        lambda = 1.0e-02        average error = 0.01898
        lambda = 2.5e-02        average error = 0.01453
        lambda = 6.3e-02        average error = 0.04365
        lambda = 1.6e-01        average error = 0.05949
        lambda = 4.0e-01        average error = 0.12835
        lambda = 1.0e+00        average error = 0.17994

Best feature set: I1, with lambda = 6.3e-04, error = 0.00185.
```

2(b) Re-fitting best model

Here we solve the same `pyomo` model, with all samples, and the better feature set with best λ value determined by Problem 2(a).

Corresponding code:

```python
# create index list for all sample points
all_point_list = list(range(better_X.shape[0]))
# create index list for features
feature_list = list(range(better_X.shape[1]))

m = pe.ConcreteModel()

m.b = pe.Var(feature_list, within=pe.Reals)
m.y_p = pe.Var(all_point_list, within=pe.Reals)
m.b_p = pe.Var(feature_list, within=pe.NonNegativeReals)
m.b_n = pe.Var(feature_list, within=pe.NonNegativeReals)

def obj_rule(m):
    return sum((m.y_p[i] - y_scaled[i]) ** 2 for i in all_point_list) + best_lambda * sum(m.b_p[j] +
    ↪  m.b_n[j] for j in feature_list)
m.obj = pe.Objective(rule=obj_rule)

def con_rule1(m, i):
    return m.y_p[i] == sum(m.b[j] * better_X[i, j] for j in feature_list)
m.con1 = pe.Constraint(all_point_list, rule=con_rule1)
def con_rule2(m, j):
    return m.b[j] == m.b_p[j] - m.b_n[j]
m.con2 = pe.Constraint(feature_list, rule=con_rule2)

solver = pe.SolverFactory('ipopt')
solver.solve(m)

# report optimal parameters and objective function
best_b = []
print("Optimal b:",end='\t')
for i in feature_list:
    best_b.append(pe.value(m.b[i]))
    print(f"{best_b[-1]:.4f}",end='\t')
print(f"\nObjective value: {pe.value(m.obj):.5f}")
```

Output:

```
Optimal b:         0.4705         -1.4921         -0.9783         1.3832         0.5128         -0.2719         0.0000
Objective value: 0.00760
```

2(c) Optimizing surrogate GLR

- **Data scaling**: the scaling should be identical in 2(a), 2(b) and 2(c): the data should be scaled for both training and optimization.
- If scaled data is used for training in 2(a), 2(b), but unscaled data is used for optimization in 2(c), then some deduction is necessary.

After fitting the regression model in Problem 2(b), now we aim to solve the following simple optimization model:

$$\min \sum_{j=0}^{m} b_j f_j(x_1, x_2)$$
$$\text{s.t. } 0 \le x_1 \le 1, 0 \le x_2 \le 1,$$

where

- f_j: features in the "better feature set" determined by Problem 2(a)
- b_j: weights of feature j determined by Problem 2(b)
- x_1, x_2: variable (scaled)

Corresponding code:

```
    # generate surrogate model surr_2c
    # dictionary for features
    terms = {}
    terms['I1'] = ['1', 'm.x1', 'm.x2', 'm.x1**2', 'm.x2**2', 'm.x1**4', 'pe.cos(m.x1)', 'm.x1*m.x2']
    terms['I2'] = ['1', 'm.x1', 'm.x2', 'm.x1**2', 'm.x1**3', 'm.x2**2', 'm.x2**3', 'm.x1*m.x2',
    ↪  'm.x1**2*m.x2', 'm.x1*m.x2**2']

    def surr_generator_pyomo(terms, b):
        """Return a string of surrogate model expression for pyomo model"""
        expr = ''
        for k in range(len(b)):
            if b[k] >= 0:
                expr = expr + '+' + str(b[k]) + '*' + terms[k]
            else:
```

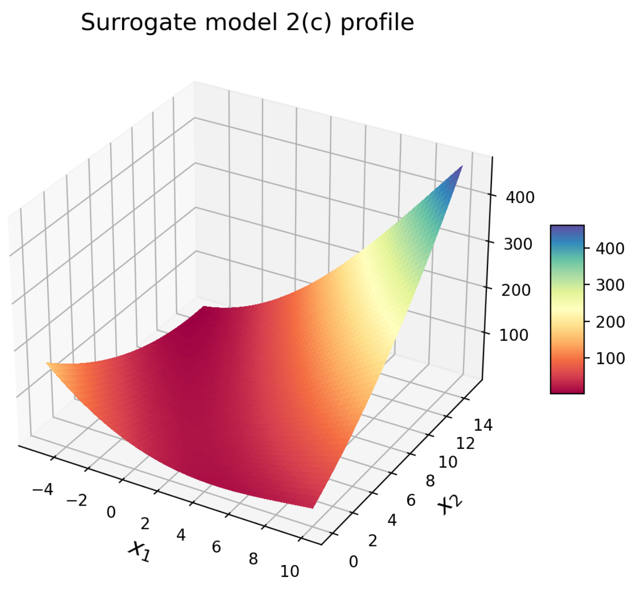
```
            expr = expr + str(b[k]) + '*' + terms[k]
        return expr

    # string for pyomo model expression
    surr_2c_str = surr_generator_pyomo(terms[better_feature_i], best_b)

    # create Pyomo model for surr_2c

    m_surr_2c = pe.ConcreteModel()
    m_surr_2c.x1 = pe.Var(bounds=(0,1),initialize=0.5)
    m_surr_2c.x2 = pe.Var(bounds=(0,1),initialize=0.5)

    def surr_2c_obj(m):
        """Transform the surrogate model expression string to python expression"""
        return eval(surr_2c_str)

    m_surr_2c.obj = pe.Objective(rule=surr_2c_obj)
    solver = pe.SolverFactory('ipopt')
    solver.solve(m_surr_2c)

    # record and print unscaled result
    result_surr_2c = {}
    result_surr_2c['x1'] = pe.value(m_surr_2c.x1) * (bounds[1,0] - bounds[0,0]) + bounds[0,0]
    result_surr_2c['x2'] = pe.value(m_surr_2c.x2) * (bounds[1,1] - bounds[0,1]) + bounds[0,1]
    result_surr_2c['obj'] = pe.value(m_surr_2c.obj) * (max(y) - min(y)) + min(y)
    print("Optimal solution of surrogate model 2(c):")
    for key, value in result_surr_2c.items():
        print(f"\t{key}:\t{value:.2f}")
```

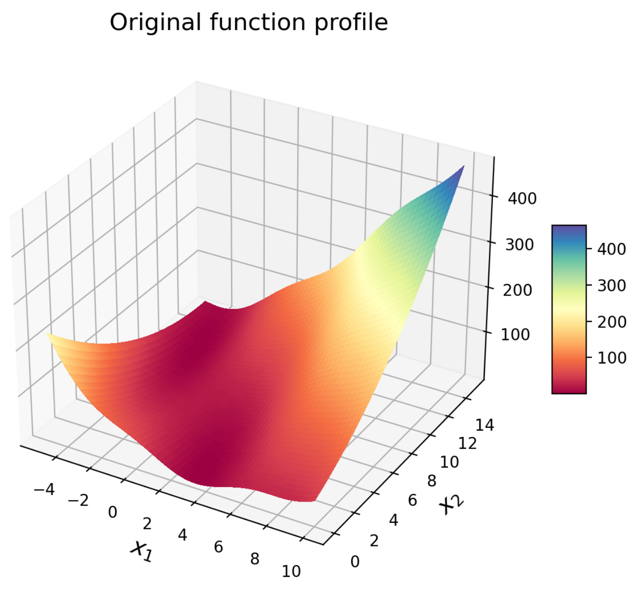
Output:

```
    Optimal solution of surrogate model 2(c):
            x1:        -5.00
            x2:        14.31
            obj:        3.08
```

Optional: Visualization of GLR surrogate model

The profile of the GLR model as well as the one of the original function is provided below to provide a visual comparison of two models (both inputs and output have been unscaled).





Problem 3 Gaussian process model (GPM)

3(a) GPM fitting

- **Workflow**: GPM fitting with 5-fold CV, aiming to find the model with the smallest objective value
- **Accuracy metric**: sum of squared errors of test set; other reasonable metrics are also acceptable
- **Discussion grading**: reasonable discussion of GPM is enough. (update: April 11)

As `sklearn` is allowed for model fitting, the formulation of GPM is not provided here.

Corresponding code:

```python
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF
from sklearn.model_selection import train_test_split

# create GPR model, with RBF kernel
gp = GPR(kernel=RBF(0.1, (1e-2, 1e2)))

cv_e = {}
cv_data = {}

# cross validation
i = 0
for train_index, test_index in kf.split(x_scaled):

    # split training and testing set
    x_train, x_test = x_scaled[train_index], x_scaled[test_index]
```

```
        y_train, y_test = y_scaled[train_index], y_scaled[test_index]

        # fit GPR model
        gpmodel = gp.fit(x_train, y_train)

        # predict
        y_pred = np.array(gpmodel.predict(x_test))

        # calculate error
        err = sum((y_pred - y_test)**2)

        # record data and error
        cv_e[i] = err
        cv_data[i] = {'x': x_train,'y': y_train}

        i += 1

    # pick the best model
    best_cv = min(cv_e, key=cv_e.get)
    best_data = cv_data[best_cv]
    best_gp = gp.fit(best_data['x'],best_data['y'])

    # calculate the average CV error
    print(f"average CV error: {sum(cv_e[i] for i in range(len(cv_e)))/len(cv_e):.4f}")
```

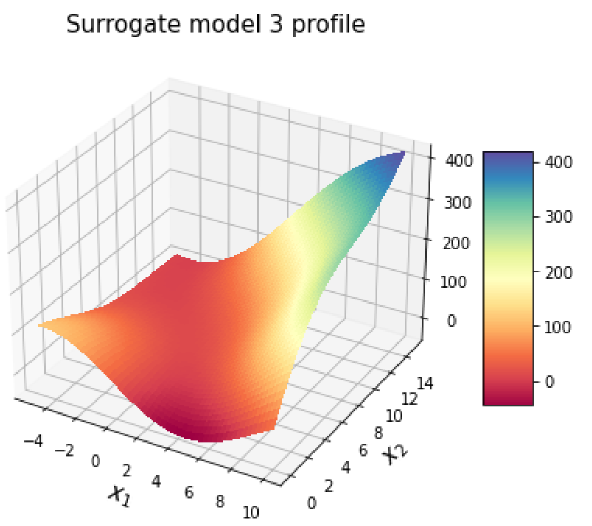
Output:

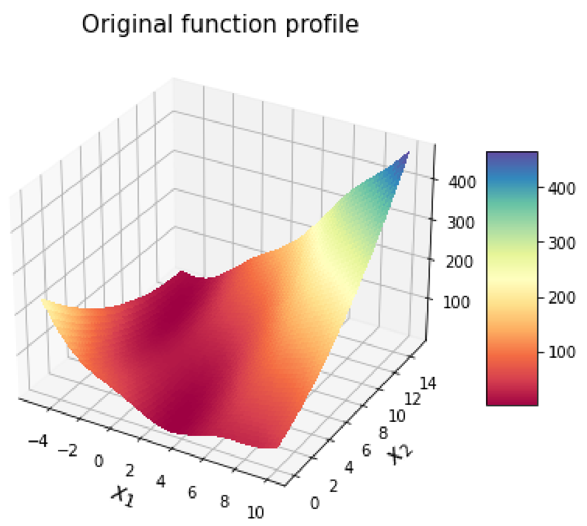
```
    average CV error: 0.0216
```

The GP model is a good approximation of the original function, as the kernel is good at capturing the nonlinearity (see the profile visualization below).

Optional: Visualization of GPM surrogate model

The profile of the GPM model as well as the one of the original function is provided below to provide a visual comparison of two models (both inputs and output have been unscaled).





3(b) Optimizing surrogate GPM

- **Results may vary**: the minimal objective value of the surrogate model may be different because of sampling, optimization method used in `scipy.optimize.minimize`
- **Bounding optional**: the Nelder-Mead method does not support bound setting, while other methods do (e.g., TNC). If the function is optimized unboundedly using Nelder-Mead, and the final result is outside of the given bound, it is acceptable.

A wrapper function is necessary to communicate between `scipy.optimize.minimize` and the GP model. It should return the predicted output value at a given input point, and the dimensions of its input and output should be adjusted according to the syntax of `scipy.optimize.minimize` and the GP model.

Corresponding code:

```python
import scipy

# define wrapper function
def gp_model(guess):
    test = np.array([[guess[0],guess[1]]])
    return best_gp.predict(test)[0]

# optimize the best GP model
initial_guess = np.array([0.5,0.5])
bnds = ((0, 1), (0, 1))
gp_optimize_result = scipy.optimize.minimize(gp_model, initial_guess, bounds=bnds, method='TNC')

# print results
result_surr_3 = {}
result_surr_3['x1'] = gp_optimize_result.x[0] * (bounds[1, 0] - bounds[0, 0]) + bounds[0, 0]
result_surr_3['x2'] = gp_optimize_result.x[1] * (bounds[1, 1] - bounds[0, 1]) + bounds[0, 1]
result_surr_3['obj'] = gp_optimize_result.fun * (max(y) - min(y)) + min(y)
print("Optimal solution of surrogate model 3:")
```

```
for key, value in result_surr_3.items():
    print(f"\t{key}:\t{value:.2f}")
```

Output:

```
Optimal solution of surrogate model 3:
        x1:        4.49
        x2:        0.00
        obj:        -49.23
```

3(c) Comparing surrogate GLR and surrogate GPM

- **Grading**: reasonable discussion about the difference of the two models is enough
- **Basic idea**: compare the minimal points of GLR, GPM with the minimal point of the original function w.r.t. inputs and output
- **Results may vary**: it is possible that either model is superior in either input or output

We can build a `pyomo` model for the original function to locate a (local) minimum: $(3.14, 1.32)$ with objective value 0.40.

model	optimal point	optimal objective value
original function	(3.14, 1.32)	0.40
GLR	(-5.00, 14.31)	3.08
GPM	(4.49, 0)	-49.23

The GLR has a better objective value than GPM, while the optimal point (input) of GPM is closer to the actual optimal point. Better results in the input space may be more meaningful, as we may further add extra sample points and re-fit the model, and being closer to the actual minimum indicates that this model is more likely to approach the actual minimum better. Therefore, in this case the GPM obtained a better solution.

Corresponding code of minimizing the original function:

```
# Pyomo model for original black-box function
m_origin = pe.ConcreteModel()
m_origin.x1 = pe.Var(bounds=(-5, 10))
m_origin.x2 = pe.Var(bounds=(0, 15))

def obj(m):
    return 10 + 10 * (1 - 1/8/np.pi) * pe.cos(m.x1) + (m.x2 - (5.1/16/(np.pi**2)) * (m.x1) ** 2 + 5
    ↪  / np.pi * m.x1 - 6)**2
m_origin.obj = pe.Objective(rule=obj)
```

```
    solver = pe.SolverFactory('ipopt')
    solver.solve(m_origin)

    result_origin = {}
    result_origin['x1'] = pe.value(m_origin.x1)
    result_origin['x2'] = pe.value(m_origin.x2)
    result_origin['obj'] = pe.value(m_origin.obj)
    print("Optimal solution of original function:")
    for key, value in result_origin.items():
        print(f"\t{key}:\t{value:.2f}")
```

Output:

```
    Optimal solution of original function:
            x1:        3.14
            x2:        1.32
            obj:        0.40
```

Problem 4 Support vector regression (SVR)

4(a) Fitting SVR & selecting C

- **Results may vary**: the values of optimal coefficients, and the minimal objective value of the surrogate model may be different because of sampling
- **Hyperparameter tuning**: no predetermined range for hyperparameter C, general grid-search or trial-and-error is enough
- **Cross validation is optional**, as it is not directly required in the assignment sheet. The example code uses 5-fold CV (similar to Problem 2), but it is also okay to simply fit the model with all data.
- **Accuracy metric**: sum of squared errors of test set; if all points are used in a single fitting, then the objective function value can be used as metric
- **Workflow**:
  1. Determine set $\mathcal{C}$ of potential C values
  2. Split training-test data for 5-fold CV
  3. For each C in $\mathcal{C}$, repeat:
     - For each fold, repeat:
       * build and solve SVR
       * calculate test error
     - calculate average error
  4. find the C with the lowest average error, report the best objective value and $\mathbf{w}$ values for this C

The formulation of SVR is given in the cited paper as Equation (3). The overall workflow is similar to the one for Problem 2(a).

Corresponding code:

```
# lambda value set
C_set = np.logspace(-4, 2, num=11)

# epsilon
epsilon = 0.001

# create index list for features
feature_list = list(range(x_scaled.shape[1]))

# create dict for average error of each C value and each model
avg_error = {}
best_models = {}

# grid-search C
for C in C_set:

    # create dictionary to record trained model for each fold
    CV_models = []
    CV_error = []

    # 5-fold CV
    for train_index, test_index in kf.split(X):

        # split training and test set
        x_train, x_test = x_scaled[train_index], x_scaled[test_index]
        y_train, y_test = y_scaled[train_index], y_scaled[test_index]

        # create index list for training points
        train_point_list = list(range(x_train.shape[0]))

        # ---------------------------------------------------------------------------------
        # Pyomo model
        m = pe.ConcreteModel()

        # define variables
        m.w = pe.Var(feature_list, within=pe.Reals)
        m.b = pe.Var(within=pe.Reals)
        m.xi_p = pe.Var(train_point_list, within=pe.NonNegativeReals)
        m.xi_n = pe.Var(train_point_list, within=pe.NonNegativeReals)

        # define objective function
        def obj_rule(m):
            return 0.5 * sum(m.w[i] ** 2 for i in feature_list) + C * sum(m.xi_p[j] + m.xi_n[j] for
            ↪  j in train_point_list)
        m.obj = pe.Objective(rule=obj_rule)

        # define constraints
        def con_rule1(m, j):
            return - m.b - sum(m.w[i] * x_train[j, i] for i in feature_list) + y_train[j] <= epsilon
            ↪  + m.xi_p[j]
        m.con1 = pe.Constraint(train_point_list, rule=con_rule1)
        def con_rule2(m, j):
            return m.b + sum(m.w[i] * x_train[j, i] for i in feature_list) - y_train[j] <= epsilon +
            ↪  m.xi_n[j]
        m.con2 = pe.Constraint(train_point_list, rule=con_rule2)

        # solve model using ipopt
        solver = pe.SolverFactory('ipopt')
        solver.solve(m)
        # ----------------------------------------------------------------------------------

        # record model for this fold
        tmp_model = {}
        # record objective value
        tmp_model['obj'] = pe.value(m.obj)
        # record parameter value
        tmp_model['w'] = np.empty((len(feature_list)))
        for i in feature_list:
            tmp_model['w'][i] = pe.value(m.w[i])
        # calculate test error
        CV_error.append(sum((np.matmul(x_test, tmp_model['w'].T) - y_test) ** 2))
        # record the model in CV_models
        CV_models.append(tmp_model)

    # calculate average error of 5-fold CV
    avg_error[C] = sum(i for i in CV_error) / len(CV_error)
    print(f'\tC = {C:.1e}\taverage error = {avg_error[C]:.5f}')

    # record the best model for a fixed C value
    best_fold = np.argmin(CV_error)
    best_models[C] = CV_models[best_fold]

# compare errors
best_C = min(avg_error, key=avg_error.get)
best_model = best_models[best_C]
print(f"\nOptimal C:\t\t\t{best_C:.1e}")
print(f"Optimal objective value: \t{best_model['obj']:.4f}")
print(f"Optimal weight: \t\t{best_model['w']}")
```

Output:

```
            C = 1.0e-04         average error = 0.56629
            C = 4.0e-04         average error = 0.56541
            C = 1.6e-03         average error = 0.56108
            C = 6.3e-03         average error = 0.54537
            C = 2.5e-02         average error = 0.48734
            C = 1.0e-01         average error = 0.40066
            C = 4.0e-01         average error = 0.28623
            C = 1.6e+00         average error = 0.64725
            C = 6.3e+00         average error = 1.10590
            C = 2.5e+01         average error = 1.17995
            C = 1.0e+02         average error = 1.20990
```

```
Optimal C:                         4.0e-01
Optimal objective value:          0.8690
Optimal weight:                  [0.362  0.2435]
```

Fitting an SVR model using sklearn should only get 50% credit. Corresponding code:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from sklearn.svm import SVR
from sklearn.model_selection import GridSearchCV as GS
from sklearn.model_selection import ShuffleSplit
import numpy as np
cg=float(0.5/(1**2))
parameters = { 'gamma' :np.linspace(1,5,20)*cg}
cost =3*np.std(y)
eps = 0.001
ss = ShuffleSplit(n_splits=5, test_size=0.25,random_state=0,train_size=0.75)
svr_rbf = GS(SVR(kernel='rbf',C=cost, epsilon = eps),param_grid=parameters,cv=ss)

model=svr_rbf.fit(x,y)

bestmodel=model.best_estimator_
y_rbf = bestmodel.predict(xx)

plt.scatter(x[:,0],y_rbf)
plt.scatter(x[:,0],y)
plt.show()
print('C is ',str(cg))
print('gamma is ',str(model.best_params_['gamma']))
print('weights are ',str(bestmodel.dual_coef_))
print('intercept is ',str(bestmodel.intercept_))
print('support vectors are ',str(bestmodel.support_vectors_))
```

4(b) Discussing SVR accuracy

- **Grading**: reasonable discussion about accuracy is enough

As we use a linear SVR here, its accuracy is worse than GLR and GRM as it is not able to capture nonlinearity.

Problem 5 PCA

5(a) How many PCs to keep

- **Scope of PCA**: PCA should only be applied to input features; if the PCA is conducted on all data in the Excel file, the results will be incorrect.
- **scaling**: necessary (update: April 11)

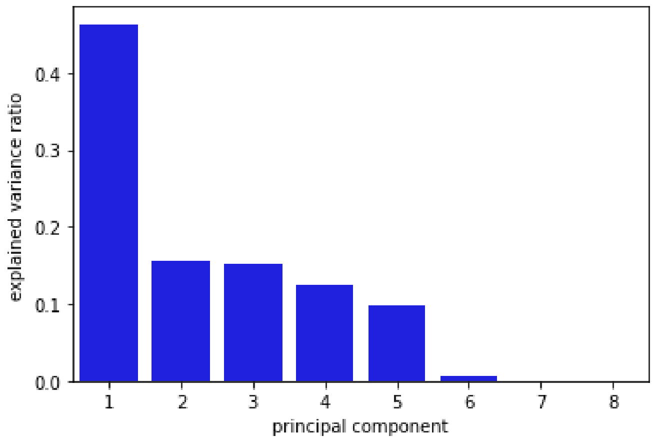
We can apply PCA on the first 8 columns in the Excel file, and check the explained variance of each PC. If the data is not scaled before PCA, some grades should be deducted for penalty. (update: April 11)

Corresponding code:

```python
import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# import data
data = pd.read_excel('ENB2012_data.xlsx',nrows=768)
# split inputs and outputs
X1=data[['Relative_Compactness','Surface_Area',
         'Wall_Area','Roof_Area','Height','Orientation',
         'Glazing_Area','Glazing_A_Distribution']]
Y1=data[['Heating_Load','Cooling_Load']]

# scale inputs
X1 = StandardScaler().fit_transform(X1)

# fit PCA model to input
pca = PCA().fit(X1)
# print explained variance ratio
for i, evr in enumerate(pca.explained_variance_ratio_):
    print(f"PC{i+1}: {evr:.2%}")

fig, ax = plt.subplots()
sns.barplot(np.arange(1,
↪   len(pca.explained_variance_ratio_)+1),pca.explained_variance_ratio_,color='blue')
ax.set_xlabel('principal component')
ax.set_ylabel('explained variance ratio')
plt.show()
```

Output:

```
PC1: 46.29%
PC2: 15.50%
PC3: 15.16%
PC4: 12.50%
PC5: 9.84%
PC6: 0.66%
PC7: 0.06%
PC8: 0.00%
```



We can see that the first 3 Principle Components explain about 75% of variance in input data. If we include the first 5 PC's we explain 99% of variance. Thus, we can choose these 5 or fewer features to reduce the dimensionality of our model. (update: April 11)

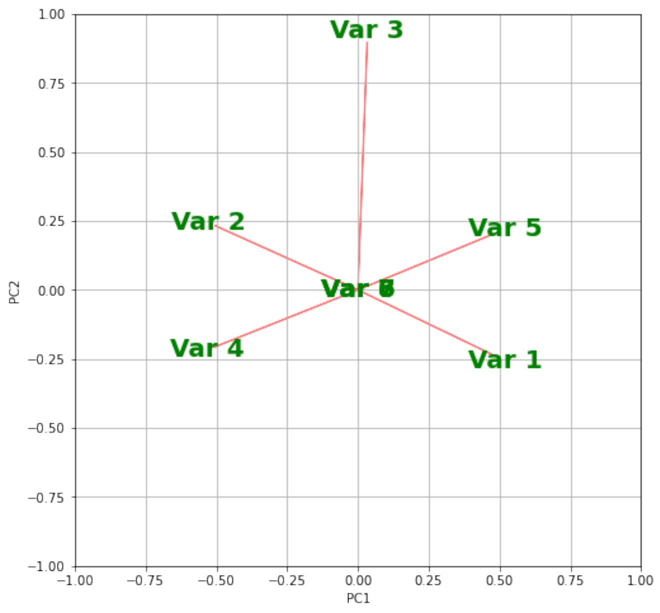
5(b) Loadings

Next we will show the load plot which tells us how much each variable contributes to each principle component.

Corresponding code:

```
# run PCA, see which variables contribute most
pc = pca.fit_transform(X1)


# define function to plot loading plot
def loading_plot(score, coeff, labels=None):
    xs = score[:, 0]
    ys = score[:, 1]
    n = coeff.shape[0]
    scalex = 1.0 / (xs.max() - xs.min())
    scaley = 1.0 / (ys.max() - ys.min())
    for i in range(n):
        plt.arrow(0, 0, coeff[i,0], coeff[i,1], color = 'r',alpha = 0.5)
        if labels is None:
            plt.text(coeff[i,0] * 1.05, coeff[i,1] * 1.05, "Var " + str(i + 1), color='g',
            ↪  ha='center', va='center', size=20, weight='bold')
        else:
            plt.text(coeff[i,0] * 1.05, coeff[i,1] * 1.05, labels[i], color='g', ha='center',
            ↪  va='center', size=20, weight='bold')

fig, ax = plt.subplots(figsize=(8, 8))
plt.xlim(-1, 1)
```

```
    plt.ylim(-1, 1)
    plt.xlabel("PC{}".format(1))
    plt.ylabel("PC{}".format(2))
    plt.grid()

    # call the function with the first two PCs
    loading_plot(pc[:,0:2], np.transpose(pca.components_[0:2, :]))
    plt.show()
```



Based on the load profiles for the 2 first PC's, Var 3 (wall area) seems to vary independently and makes up a large part of PC 1. PC 2 is comprised of fairly equal parts of Vars 1, 2, 4, and 5 (relative compactness, surface area, roof area, height). (update: April 11)

5(c) Discussion

- **Grading**: reasonable discussion about PCA is enough

PCA is useful for exploring our data set and doing some basic feature engineering. Much of the variance in our input data can be explained in lower dimensional representations through linear PCA.