Demonstration of Bayesian Optimization on a L_1^ϵ Support Vector Regression's hyperparamters

Gabriel Morales Ruiz
M.Sc. in Data Science
Instituto Tecnológico y de Estudios Superiores de Occidente
Tlaquepaque, Jalisco
gbmrls@gmail.com

***Abstract*—The purpose of this document is to compare the R^2 score of an L_1^ϵ Support Vector Regression, whose hyperparamters (ϵ, c and γ) have been optimized, vs an Ordinary Least Squares regression. Both models are trained with the Boston Housing Dataset.**

***Index Terms*—svr, soft margin, ols, bayesian optimization**

I. Introduction

A. Ordinary Least Squares Regression

The ordinary least squares regression problem consists of finding a linear combination of features X that best describe the desired output y. For $i = 1, \ldots, n$ the mean of the conditional distribution of y_i is given by a feature vector x_i, in the form of

$$y_i = x_i^T\theta + \epsilon_i. \tag{1}$$

θ and x_i are vectors with size $k \times 1$, and ϵ_i is a vector of independent and identically distributed variables such that ϵ_i $N(0, \sigma^2)$. The problem's maximum likelihood function is:

$$L(Y \mid X, \theta, \sigma^2) \propto (\sigma^2)^{-n/2} e^{-\frac{1}{2\sigma^2}(Y - X\theta)^T(Y - X\theta)} \tag{2}$$

To maximize it, the expression to minimize is:

$$\min_\theta \frac{1}{2}(Y - X\theta)^T(Y - X\theta) \tag{3}$$

B. L1 Soft Margin Support Vector Regression

In support vector regression, the input space is mapped into the feature space, where an optimal hyperplane is given by

$$f(x) = w^T\varphi(x) + b, \tag{4}$$

where $\varphi : X \to \mathcal{F}$ is a function that makes each input point x correspond to a point in $\mathcal{F}$, where $\mathcal{F}$ is a Hilbert space. As seen from equation (3), OLS utilizes the squared residuals to fit the parameters θ. However, large residuals cause by outliers may worsen the accuracy significantly. [1].

SVR uses a piecewise linear function to counter this, in which a hyperparameter ϵ called the margin lets errors that are less or equal to it be 0, and errors larger than it be error$-\epsilon$. Any prediction inside the radius of ϵ counts as a correct prediction. The problem to solve

$$\begin{aligned}
\min_{w,b,\xi,\xi^*} \mathcal{P}_\epsilon(w, b, \xi, \xi^*) &= \frac{1}{2}w^Tw + c\sum_{k=1}^{N}(\xi_k + \xi_k^*) \\
\text{s.t. } & y_k - w^T\varphi(x_k) - b \le \epsilon + \xi_k, k = 1, ..., N \\
& w^T\varphi(x_k) + b - y_k \le \epsilon + \xi_k^*, k = 1, ..., N \\
& \xi_k, \xi_k^* \ge 0, k = 1, ..., N
\end{aligned} \tag{5}$$

The Lagrangian for this problem is given by

$$\begin{aligned}
L(w, b, \xi, \xi^*; \alpha, \alpha^*, \eta, \eta^*) = & \frac{1}{2}w^Tw + c\sum_{k=1}^{N}(\xi_k + \xi_k^*) \\
& - \sum_{k=1}^{N}\alpha_k\{\epsilon + \xi_k - y_k + w^T\varphi(x_k) + b\} \\
& - \sum_{k=1}^{N}\alpha_k^*\{\epsilon + \xi_k^* + y_k - w^T\varphi(x_k) - b\} \\
& - \sum_{k=1}^{N}\eta_k\xi_k - \sum_{k=1}^{N}\eta_k^*\xi_k^* \\
\text{s.t. } & \alpha, \alpha^*, \eta, \eta^* \succeq 0
\end{aligned} \tag{6}$$

where the Lagrange multipliers must be greater than or equal to zero to not disturb the inequalities.

The stationary conditions derived from the problem's Lagragian are the following:

$$\begin{aligned}
\nabla_w L = w - \sum_{k=1}^{N}\alpha_k\varphi(x_k) + \sum_{k=1}^{N}\alpha_k^*\varphi(x_k) = 0 \\
\Rightarrow w = \sum_{k=1}^{N}(\alpha_k - \alpha_k^*)\varphi(x_k)
\end{aligned} \tag{7}$$

$$\frac{\partial L}{\partial b} = \sum_{k=1}^{N}\alpha_k - \sum_{k=1}^{N}\alpha_k^* = 0 \Rightarrow \sum_{k=1}^{N}(\alpha_k - \alpha_k^*) = 0 \tag{8}$$

$$\begin{aligned}
\frac{\partial L}{\partial \xi_k} = c - \alpha_k - \eta_k = 0, \quad k = 1, ..., N \\
c - \alpha_k = \eta_k, \quad \eta_k \ge 0 \; \forall \; k \Rightarrow c - \alpha_k \ge 0 \therefore \; \alpha_k \le c
\end{aligned} \tag{9}$$

$$\frac{\partial L}{\partial \xi_k^*} = c - \alpha_k^* - \eta_k^*, \quad k = 1, ..., N$$
$$c - \alpha_k^* = \eta_k^*, \quad \eta_k^* \geq 0 \;\forall\; k \Rightarrow c - \alpha_k^* \geq 0 \therefore\; \alpha_k^* \leq c \tag{10}$$

Wolfe's dual problem is obtained from substituting equations (7), (8), (9) and (10) back into (6), and its solution is to find the α and α^* that maximize its output.

$$\begin{aligned} D(\alpha, \alpha^*) = &-\frac{1}{2}\sum_{k,l=1}^{N}(\alpha_k - \alpha_k^*)(\alpha_l - \alpha_l^*)\varphi(x_k)^T\varphi(x_l) \\ &+ \sum_{k=1}^{N}(\alpha_k - \alpha_k^*)y_k - \epsilon\sum_{k=1}^{N}(\alpha_k + \alpha_k^*) \\ \text{s.t.}\quad & 0 \preceq \alpha \preceq c, \quad 0 \preceq \alpha^* \preceq c \\ & \sum_{k=1}^{N}(\alpha_k - \alpha_k^*) = 0 \end{aligned} \tag{11}$$

When $0 < \alpha_k < c$ or $0 < \alpha_k^* < c$

$$\eta_k\xi_k = (c - \alpha_k)\xi_k = 0,\; \alpha_k < c \;\therefore\; c - \alpha_k > 0 \Rightarrow \xi_k = 0$$
$$\eta_k^*\xi_k^* = (c - \alpha_k^*)\xi_k^* = 0,\; \alpha_k^* < c \;\therefore\; c - \alpha_k^* > 0 \Rightarrow \xi_k^* = 0$$

Thus, the bias term is defined as

$$\begin{aligned} b &= y_k - w^T\varphi(x_k) - \epsilon, \quad 0 < \alpha_k < c \\ b &= y_k - w^T\varphi(x_k) + \epsilon, \quad 0 < \alpha_k^* < c \end{aligned} \tag{12}$$

Parameter b is obtained from averaging all of these possible solutions.

C. Bayesian Optimization

Bayesian optimization is an approach that utilizes Bayes' Theorem to direct a search to find either the minimum or maximum of an objective function. This is accomplished by performing a regression of the low-sampled objective function with a Gaussian process, and then using this surrogate function to direct the search. [2]

II. Implementation

The key performance indicator for the tests is the R^2 score. This is measured with the same Python library for both models (sklearn).

A. Treating the dataset

In order to reduce noise in the comparison, the same treated dataset is used for both models (Boston Housing Dataset [4]). The steps that go into engineering the features and output is:

1) All samples are shuffled.
2) The features and the output are scaled using sklearn's StandardScaler, which centers their mean at zero and scales them by dividing each variable by its standard deviation.
3) The dataset is split in two parts. 40% is used for training the models and 60% is used to test them.

B. OLS Regression

The OLS Regression was implemented using the statsmodels package [6]. This package was used to give the linear regression a better chance at performing better, to give it a fighting chance against the optimized SVR.

The R^2 for the samples used to train it was 0.75, and the R^2 for the test samples was 0.70.

C. SVR with Bayesian Optimization

The SVR and Bayesian Optimization codes were written by me, as requirement of the course's rubric.

1) SVR: The dual from equation (11) was modified to ease the burden on the solver.

$$\begin{aligned} \beta &= \alpha - \alpha^* \\ |\beta| &= \alpha + \alpha^* \end{aligned} \tag{13}$$

If we rewrite equation (11) in matrix form, and substitute equation (13) into it, we get:

$$\begin{aligned} D(\beta) = &-\frac{1}{2}\beta^T K\beta + \beta^T y + \epsilon|\beta|^T 1_v \\ \text{s.t.}\quad & -c \preceq \beta \preceq c, \\ & \beta^T 1_v = 0 \end{aligned} \tag{14}$$

A Gaussian kernel was used for the model, which adds the hyperparameter γ into the list of things to optimize, and ECOS [5] was used to solve the expression.

2) Bayesian Optimization: To properly perform a Bayesian Optimization, a Gaussian process regression class was created. This is very similar to the Gaussian kernel used for the SVR. The Gaussian process assumes that for a small change of x there will be a small change in $f(x)$; in other words, it assumes that the function is smooth.

The Bayesian Optimization algorithm fits the Gaussian process regression on the training data (X, y), and uses the regression to look for the next optimal point to sample on the objective function. [3]

The objective function is a $f(c, \epsilon, \gamma)$, which returns the R^2 score and takes a long time to compute. To work with this objective function, the SVR needs to be fitted with some parameters to have some prior information.

3) Integration: I did not have enough time to research on how to perform a multivariate Gaussian process. My solution to this was to concatenate the optimizations, which is very slow, but yields similar results to using a precoded Python package.

1) Objective 1 was to optimize the R^2 score by modifying c, given ϵ and γ.
2) Objective 2 was to optimize the R^2 score by modifying ϵ given γ, wherein for every ϵ c was optimized with Objective 1.
3) Objective 3 was to optimize the R^2 score by modifying γ, wherein for every γ ϵ was optimized with Objective 2.

The algorithm got the following values to maximize the R^2 score:

- $c = 20$
- $\epsilon = 0.04244$
- $\gamma = 1.32500$

The R^2 that was obtained from this optimized model was 0.81.

III. Conclusions

The optimized SVR model certainly got a better result, but due to my lack of expertise the concatenated solution takes about one and a half hours to run. The OLS model could also be optimized upon, by removing not statistically significant features. These were kept to minimize any differences between the samples used to train both models.

References

[1] Shigeo Abe. *Support Vector Machines for Pattern Classification, 2 Ed.* Springer-Verlag London, 2010.

[2] C.E. Rasmussen, and C.K.I. Williams. *Gaussian Processes for Machine Learning* MIT Press, 2006.

[3] J. Brownlee. *How to Implement Bayesian Optimization from Scratch in Python.* Machine Learning Mastery, 21-Aug-2020. [Online]. Available: https://machinelearningmastery.com/what-is-bayesian-optimization/. [Accessed: 26-Apr-2021].

[4] D. Harrison and D.L. Rubinfeld. *Hedonic housing prices and the demand for clean air.* Journal of Environmental Economics and Management, 1978.

[5] A. Domahidi, E. Chu and S. Boyd, *ECOS: An SOCP solver for embedded systems,* European Control Conference (ECC), 2013, pp. 3071-3076,

[6] S. Seabold, and J. Perktold. “Statsmodels: Econometric and statistical modeling with python.” Proceedings of the 9th Python in Science Conference. 2010.

A. L_1^ϵ SVR Code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 21 17:30:02 2021

@author: Gabriel A. Morales Ruiz
"""

import numpy as np
import cvxpy as cp
from sklearn import metrics

class L1e_SVR :

    def __init__(self, c = 1, e = 1, k_type = "rbf", sigma = 2) :
        """
        Constructor for SVR object.

        Parameters
        ----------
        c : float, optional
            Regularization hyperparameter. Must be greater than 0.
            The default is 1.
        e : float, optional
            Epsilon hyperparameter: Radius of the margin.
            Must be greater than 0. The default is 1.
        k_type : str, optional
            Kernel type. Options are "linear" and "rbf".
            The default is "rbf".
        sigma : float, optional
            Hyperparameter used on rbf kernel. The default is 2.

        Returns
        -------
        None.

        """
        self.c = c
        self.e = e
        self.k_type = k_type
        self.sigma = sigma

        self.X = None
        self.b = None
        self.y = None
        self.beta = None

    def Kernel(self, xk, xl) :
        """
        phi(xk)phi(xl)^T
        Can be either linear or rbf
        Parameters
        ----------
        xk : np.matrix
```

```
            Feature vector (new features when using to predict)
        xl : np.matrix
            Feature vector (fitted X)

        Raises
        ------
        AssertionError
            Error when kernel type is unknown

        Returns
        -------
        np.matrix
            Result of operation

        """
        if self.k_type == "linear" :
            return xk @ xl.T
        elif self.k_type == "rbf" :
            N1 = xk.shape[0]
            N2 = xl.shape[0]
            kernel = np.zeros((N1, N2))
            for k in range(N1) :
                for l in range(N2) :
                    kernel[k, l] = np.exp(\
                        - np.linalg.norm(xk[k, :] - xl[l, :])**2 / \
                            (2*self.sigma**2))
            return kernel
        else :
            raise AssertionError("Unknown kernel type.")


    def fit(self, X, y) :
        """
        Trains the SVR with the input parameters

        Parameters
        ----------
        X : np.matrix
            Features
        y : np.matrix
            Known output

        Returns
        -------
        None.

        """
        K = self.Kernel(X, X)

        N = X.shape[0]
        beta = cp.Variable((N, 1))
        v1 = np.ones((N, 1))
        ev = v1*self.e

        dual = cp.quad_form(beta, K)/2 + ev.T @ cp.abs(beta) - y.T @ beta
```

```
        constraints = [v1.T @ beta == 0,
                       beta <= self.c,
                       beta >= -self.c]
        cp.Problem(cp.Minimize(dual), constraints).solve(solver = "ECOS")
        beta = np.matrix(beta.value)

        # Support vectors
        sv = abs(beta) > 1e-5
        self.beta = beta[sv].T
        n_sv = self.beta.shape[0]
        self.X = X[np.repeat(sv, X.shape[1], axis=1)].reshape(n_sv, X.shape[1])

        # Compute b
        sb = np.logical_and( abs(beta) > 1e-5, abs(beta) < self.c )
        beta_sb = beta[sb].T
        n_sb = beta_sb.shape[0]
        y_sb = y[sb].T
        K_sb = K[sb*sb.T].reshape(n_sb, n_sb)
        e_sb = np.sign(beta_sb)*self.e
        self.b = np.mean(y_sb - (K_sb*beta_sb) + e_sb)

    def predict(self, X_new) :
        """
        Uses the fitted model to predict an output with input features

        Parameters
        ----------
        X_new : np.matrix
            Features

        Returns
        -------
        np.matrix
            Prediction based on input.

        """
        K = self.Kernel(X_new, self.X)
        return (sum(np.multiply(self.beta, K.T)) + self.b).T

    def mse(self, X_new, y_new) :
        """
        Calculates the mean squared error

        Parameters
        ----------
        X_new : np.matrix
            Input to predict.
        y_new : np.matrix
            Correct result

        Returns
        -------
        float
            Mean squared error of the prediction - the real value.

        """
```

```python
        y_pred = self.predict(X_new)
        mse = np.square(y_pred - y_new).mean(axis=0)
        return mse[0, 0]

    def score(self, X_new, y_new) :
        """
        Calculates R^2 score

        Parameters
        ----------
        X_new : np.matrix
            Input to predict
        y_new : TYPE
            Correct result

        Returns
        -------
        float
            R^2 score.

        """
        y_pred = self.predict(X_new)
        return metrics.r2_score(y_pred, y_new)
```

B. Bayesian Optimization Code

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 29 18:56:03 2021

@author: Gabriel A. Morales Ruiz
"""

import numpy as np
from scipy.stats import norm

class GaussianProcessRegression :
    def __init__(self, l) :
        """
        Constructor.

        Parameters
        ----------
        l : float
            Parameter used on the Gaussian Process.

        Returns
        -------
        None.

        """
        self.l = l
        self.k = None
        self.X = None
        self.y = None

    def K(self, xk, xl) :
        """
        Gaussian process' kernel function

        Parameters
        ----------
        xk : np.matrix
            Feature vector.
        xl : np.matrix
            Feature vector.

        Returns
        -------
        np.matrix
            Output kernel.

        """
        v1k = np.ones((1, xk.shape[0]))
        v1l = np.ones((1, xl.shape[0]))
        Xk = xk*v1l;
        Xl = (xl*v1k).T
        return np.exp(-np.square(Xk - Xl)/(2*self.l))

    def fit(self, X, y) :
```

```
        """
        Stores inputs and outputs, and calculates the kernel.

        Parameters
        ----------
        X : np.matrix
            Input features.
        y : TYPE
            Expected output.

        Returns
        -------
        None.

        """
        self.X = X
        self.y = y
        self.k = self.K(X, X)

    def predict(self, X_test, return_std = False) :
        """
        Uses the fitted information to predict an output based on new features.

        Parameters
        ----------
        X_test : np.matrix
            New features to used on prediction
        return_std : boolean, optional
            Use if you want this function to calculate and return
            the prediction's standard deviation. The default is False.

        Returns
        -------
        mu : float
            Mean value of prediction.
        s : float
            Standard deviation of prediction.

        """
        # Mean
        L = np.linalg.cholesky(self.k + 1e-6*np.eye(self.k.shape[0]))
        Lk = np.linalg.solve(L, self.K(self.X, X_test))
        mu = np.dot(Lk.T, np.linalg.solve(L, self.y))

        # Variance
        if return_std :
            kss = self.K(X_test, X_test)
            s2 = np.diag(kss) - np.sum(np.square(Lk), axis=0)
            s = np.sqrt(s2)
            return mu, s
        else : return mu

class GaussianOptimization :
    def __init__(self, objective, X, y, l=1, minimize = True, pool=100) :
        """
        Constructor
```

```python
        Parameters
        ----------
        objective : function pointer
            Pointer to .
        X : np.matrix
            Initial samples' input.
        y : np.matrix
            Initial samples' output.
        l : float, optional
            Gaussian process' hyperparameter. The default is 1.
        minimize : boolean, optional
            If false, then the optimization will maximize. The default is True.
        pool : int, optional
            Amount of points to use when calculating the next optimal point.
            The default is 100.

        Returns
        -------
        None.

        """
        self.gpr = GaussianProcessRegression(l)
        self.X = X
        self.y = y
        self.objective = objective
        self.minimize = minimize
        self.pool = pool
        self.gpr.fit(X, y)

    def surrogate(self, X) :
        """
        Uses the gaussian regression's function to simulate the objective
        function's output.

        Parameters
        ----------
        X : np.matrix
            Input to gaussian regression.

        Returns
        -------
        np.matrix
            Predicted value.

        """
        return self.gpr.predict(X, True)

    def acquire(self) :
        """
        Creates samples and runs them through the gaussian regression
        prediction. Then, this function uses current data to calculate whether
        one of those points has a higher chance of yielding a better result.

        Returns
        -------
```

```
        float
            Optimal point to use on objective function.

        """
        X_new = np.random.uniform(min(self.X), max(self.X), self.pool)
        # X_new = np.random.random(self.pool)
        X_new = X_new.reshape(self.pool, 1)


        y_pred, _ = self.surrogate(self.X)
        if self.minimize : best = min(y_pred)
        else :             best = max(y_pred)

        mu, std = self.surrogate(X_new)
        mu = mu[:, 0]

        probs = norm.cdf((mu - best)/(std + 1e-9))

        ix = np.argmax(probs)
        return X_new[ix, 0]

    def optimize(self, iters = 100) :
        """
        Runs the acquire function and evalutes on objective function with the
        purpose to find the max/min value.

        Parameters
        ----------
        iters : TYPE, optional
            DESCRIPTION. The default is 100.

        Returns
        -------
        None.

        """
        for i in range(iters) :
            print("  Generation " + str(i))
            x = self.acquire()
            y = self.objective(x)

            mu, _ = self.surrogate(np.matrix(x))
            self.X = np.vstack((self.X, [[x]]))
            self.y = np.vstack((self.y, [[y]]))

            self.gpr.fit(self.X, self.y)
```

C. Integration Code

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 29 18:54:37 2021

@author: Gabriel Alejandro Morales Ruiz
"""

from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from L1eSVR import L1e_SVR
import numpy as np
import matplotlib.pyplot as plt
from GaussianOptimization import GaussianOptimization
from sklearn import preprocessing
from sklearn.utils import check_random_state

# Global variables
e_ = 0.2
sigma_ = 2
results = []

def plot(X, y, model):
    """
    Simple function to plot the model and objective function plots (scatter)

    Parameters
    ----------
    X : np.list
        Objective function samples' input.
    y : np.list
        Objective function samples' output.
    model : TYPE
        Gaussian Optimization model.

    Returns
    -------
    None.

    """
    plt.scatter(X, y) # Scatter plot of real pairs
    # Sample surrogate function
    Xsamples  = np.linspace(min(X), max(X), 200).reshape(200, 1)
    ysamples, _ = model.surrogate(Xsamples)
    plt.plot(Xsamples, ysamples) # Continuous plot of surrogate function
    plt.show()

#%% Database
rng = check_random_state(0) # Seed for reproducibility

boston = load_boston()

# Shuffle
perm = rng.permutation(boston.target.size)
boston.data = boston.data[perm]
```

```python
boston.target = boston.target[perm]

# Assign
y = boston.target
y=np.reshape(y,(len(boston.target),1))
X = boston.data

# Standarize
scaler = preprocessing.StandardScaler().fit(X)
X_scaled = scaler.transform(X)
scaler = preprocessing.StandardScaler().fit(y)
y_scaled = scaler.transform(y)

# Split
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
                                                    y_scaled,
                                                    test_size=0.6,
                                                    random_state = 6)
n_train = X_train.shape[0]

#%% SVR without hyperparameter optimization

svr_base = L1e_SVR(k_type="rbf", c=1, e=e_)
svr_base.fit(X_train, y_train)
r2_base = svr_base.score(X_test, y_test)
print("Base model's score: " + str(r2_base))


#%% Optimization functions

# R^2 score vs C normalization hyperparameter.
def objective1(c, X=X_train, y=y_train, X_test=X_test, y_test=y_test) :
    global e_
    global sigma_
    global results
    svr = L1e_SVR(k_type="rbf", c=c, e=e_)
    svr.fit(X, y)
    r2 = svr.score(X_test, y_test)
    results.append([c, e_, sigma_, r2])
    return r2

def sample_objective1() :
    C = np.linspace(1, 20, 5)
    r2 = []
    for c in C :
        #print("Using c=" + str(c))
        r2.append(objective1(c))
    r2 = np.asarray(r2)
    C_mat = C.reshape(len(C), 1)
    r2_mat = r2.reshape(len(r2), 1)
    return C_mat, r2_mat

obj1_best_c = None

# R^2 score vs epsilon (margin) hyperparameter.
def objective2(e, X=X_train, y=y_train, X_test=X_test, y_test=y_test) :
```

```
    global e_
    global sigma_
    global obj1_best_c
    e_ = e
    C_mat, r2_mat = sample_objective1()
    model_c = GaussianOptimization(objective1,
                                   C_mat,
                                   r2_mat,
                                   minimize = False,
                                   l = 3)
    model_c.optimize(iters=10)
    ix = np.argmax(model_c.y)
    obj1_best_c = model_c.X[ix]
    svr = L1e_SVR(k_type="rbf", c = obj1_best_c, e=e_, sigma=sigma_)
    svr.fit(X, y)
    r2 = svr.score(X_test, y_test)
    return r2

def sample_objective2() :
    E = np.linspace(0.01, 2, 5)
    r2 = []
    for e in E :
        #print("Using e=" + str(e))
        r2.append(objective2(e))
    r2 = np.asarray(r2)
    E_mat = E.reshape(len(E), 1)
    r2_mat = r2.reshape(len(r2), 1)
    return E_mat, r2_mat

# R^2 score vs kernel (sigma) hyperparameter.
def objective3(sigma, X=X_train, y=y_train, X_test=X_test, y_test=y_test) :
    global obj1_best_c
    global sigma_
    sigma_ = sigma
    E_mat, r2_mat = sample_objective2()
    model_e = GaussianOptimization(objective2,
                                   E_mat,
                                   r2_mat,
                                   minimize = False,
                                   l = 0.5)
    model_e.optimize(iters=7)
    ix = np.argmax(model_e.y)
    svr = L1e_SVR(k_type="rbf", c=obj1_best_c, e=model_e.X[ix], sigma=sigma_)
    svr.fit(X, y)
    r2 = svr.score(X_test, y_test)
    return r2

def sample_objective3() :
    S = np.linspace(0.1, 5, 5)
    r2 = []
    for s in S :
        print("Using s=" + str(s))
        r2.append(objective3(s))
    r2 = np.asarray(r2)
    S_mat = S.reshape(len(S), 1)
    r2_mat = r2.reshape(len(r2), 1)
```

```python
    return S_mat, r2_mat

#%% Optimization process
S_mat, r2_mat = sample_objective3()
model = GaussianOptimization(objective3,
                             S_mat,
                             r2_mat,
                             minimize = False,
                             l=1)
model.optimize(iters=10)

#%% Results extraction
res_mat = np.matrix(results)
c_list = np.array(res_mat[:, 0]).ravel()
e_list = np.array(res_mat[:, 1]).ravel()
s_list = np.array(res_mat[:, 2]).ravel()
r2_list = np.array(res_mat[:, 3]).ravel()

ix = np.argmax(r2_list)
print("Best result:")
print("  c = " +  str(c_list[ix]))
print("  e = " +  str(e_list[ix]))
print("  sigma = " +  str(s_list[ix]))
print("  r^2 = " +  str(r2_list[ix]))


#%% OLS Regression
import statsmodels.api as sm
import pandas as pd
from sklearn.metrics import r2_score
data = pd.DataFrame(np.hstack((X_train, y_train)))
X_ols = pd.DataFrame(X_train)
y_ols = pd.DataFrame(y_train)
mlr = sm.OLS(y_ols, X_ols).fit()

X_ols_test = pd.DataFrame(X_test)
y_ols_test = pd.DataFrame(y_test)

sm.add_constant(X_ols_test)
y_ols_pred = mlr.predict(X_ols_test)
r2_ols = r2_score(y_test, y_ols_pred)
print("Ordinary Least Squares Regression")
print(" r2: " + str(r2_ols))
```