ELL409: MACHINE LEARNING

ASSIGNMENT 3

Implementation of Support Vector Regression (SVR) using CVXOPT and using sklearn

HARSH KUMAR
2017MT10729

Dataset is Bosten House Pricing Data, consisting of 506 rows and 13 features and a class containing the Median value of owner-occupied homes to be predicted later.
First will look after SVR implementation using CVXOPT

Implementation of SVR using CVXOPT:
We need to solve a dual problem in this, as discussed in the class.

$$\widetilde{L}(\mathbf{a}, \widehat{\mathbf{a}}) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(a_n - \widehat{a}_n)(a_m - \widehat{a}_m)k(\mathbf{x}_n, \mathbf{x}_m) - \epsilon\sum_{n=1}^{N}(a_n + \widehat{a}_n) + \sum_{n=1}^{N}(a_n - \widehat{a}_n)t_n$$

We need to maximize this equation with respect to constraints

$$\sum_{n=1}^{N}(a_n - \widehat{a}_n) = 0$$

$$0 \leqslant a_n \leqslant C$$
$$0 \leqslant \widehat{a}_n \leqslant C$$

Where C and epsilon are hyperparameters.
Using the value of “a” and “a^” found we can find find the value of y(x), which is as follows:

$$y(\mathbf{x}) = \sum_{n=1}^{N}(a_n - \widehat{a}_n)k(\mathbf{x}, \mathbf{x}_n) + b$$

Where b is

$$\begin{aligned} b &= t_n - \epsilon - \mathbf{w}^{\mathrm{T}}\phi(\mathbf{x}_n) \\ &= t_n - \epsilon - \sum_{m=1}^{N}(a_m - \widehat{a}_m)k(\mathbf{x}_n, \mathbf{x}_m) \end{aligned}$$

Then we took the average of all the b for which points are support vectors.

The above dual problem is solved using a QP Solver, whose standard form is given below:

$$\begin{aligned} \min_{x} \quad & \tfrac{1}{2}x^\top P x + q^\top x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

We can get the value of x using the following equations:

```
sol = solvers.qp(P,q,G,h,A,b)

sol['x']
```

Hence we need to convert our dual problem in the above form. So, took x as a matrix of [a, a^] and then changes each equation accordingly, which is shown below.

- First, data normalization is done.

```
# normalisation of data
data = np.array(data)
data -= np.mean(data, axis=0)
data /= np.std(data, axis=0)
```

- The different kernel function is introduced as:

```
# define the Kernal Function for it is RBF
def kernal_rbf(x, y, gamma):
  k = math.exp((-1)*gamma*(LA.norm(x-y)*(LA.norm(x-y))))
  return k
```

```
# define the polynomial kernal function, d id the degree of freedom pf polynomial and cons is the constant term
def kernal_polynomial(x, y, constant, d):
  k = pow(np.dot(x, y)+constant, d)
  return k
```

```
# define a linear kernal function
def kernal_linear(x, y, cons):
  k = np.dot(x, y)+cons
  return k
```

- Then a kernel matrix is introduced for each kernel function such that K = K[Xi, Xj] for 0<=i, j<=n, where n is the number of data vectors.

```
def kernal_matrix_rbf(X, gamma):
  n = len(X);
  k = np.zeros([n, n])
  for i in range(len(X)):
    for j in range(len(X)):
      k[i][j] = kernal_rbf(X[i], X[j], gamma)
  return k
```

```
def kernal_matrix_polynomial(X, constant, d):
  n = len(X);
  k = np.zeros([n, n])
  for i in range(len(X)):
    for j in range(len(X)):
      k[i][j] = kernal_polynomial(X[i], X[j], constant, d)
  return k
```

```python
def kernal_matrix_linear(X, cons):
  n = len(X);
  k = np.zeros([n, n])
  for i in range(len(X)):
    for j in range(len(X)):
      k[i][j] = kernal_linear(X[i], X[j], cons)
  return k
```

- The P matrix will be equal to [[k, -k],[-k,k]], which is implemented as below for each of the kernel functions

```python
# calculate P matrix for rbf
def P_Matrix_rbf(X, gamma):
  k = kernal_matrix_rbf(X,gamma)
  neg_k = (-1)*k
  temp1 = np.concatenate((k, neg_k))
  temp2 = np.concatenate((neg_k, k))
  P = np.concatenate((temp1, temp2), axis = 1)
  return P
```

```python
# calculate P matrix for linear
def P_Matrix_linear(X, cons):
  k = kernal_matrix_linear(X, cons)
  neg_k = (-1)*k
  temp1 = np.concatenate((k, neg_k))
  temp2 = np.concatenate((neg_k, k))
  P = np.concatenate((temp1, temp2), axis = 1)
  return P
```

```python
# calculate P matrix for polynomial
def P_Matrix_polynomial(X, constant, d):
  k = kernal_matrix_polynomial(X, constant, d)
  neg_k = (-1)*k
  temp1 = np.concatenate((k, neg_k))
  temp2 = np.concatenate((neg_k, k))
  P = np.concatenate((temp1, temp2), axis = 1)
  return P
```

- Q matrix is implemented as follows:

```
# calculate q matrix
def q_matrix(y, elp):
  temp1 = -y+elp
  temp2 = y+elp
  q = np.concatenate((temp1, temp2))
  return q
```

- G Matrix:

```
# calculate G matrix
def G_matrix(n):
  temp1 = np.identity(2*n)
  temp2 = (-1.0)*temp1
  g = np.concatenate((temp1, temp2))
  return g
```

- H Matrix:

```
# calculate h matrix
def h_matrix(C, n):
  temp1 = np.ones(2*n)
  temp2 = np.zeros(2*n)
  h = np.concatenate((temp1, temp2))
  return h
```

- A matrix:

```
# calculate A matrix
def A_matrix(n):
  temp1 = np.ones(n)
  temp2 = (-1.0)*temp1
  temp = np.concatenate((temp1, temp2))
  temp = temp.reshape(1,-1)
  return temp
```

- B matrix:

```
# calculate b matrix
def b_matrix():
  b = np.zeros(1)
  return b
```

- Now, the prediction function for different kernel function is implemented as follows:

```
# prediction of y for rbf
def Y_predicted_rbf(X, n, lamda_1, lamda_2, X_train, b, gamma):
  total = 0
  for i in range(n):
    total = total+(lamda_1[i][0]-lamda_2[i][0])*kernal_rbf(X, X_train[i], gamma)
  y = total+b
  return y
```

```
# prediction of y for linear
def Y_predicted_linear(X, n, lamda_1, lamda_2, X_train, b, cons):
  total = 0
  for i in range(n):
    total = total+(lamda_1[i][0]-lamda_2[i][0])*kernal_linear(X, X_train[i], cons)
  y = total+b
  return y
```

```
# prediction of y for polynomial
def Y_predicted_polynomial(X, n, lamda_1, lamda_2, X_train, b, constant, d):
  total = 0
  for i in range(n):
    total = total+(lamda_1[i][0]-lamda_2[i][0])*kernal_polynomial(X, X_train[i], constant, d)
  y = total+b
  return y
```

- Math Square Error is calculated as follows:

```
# calculation of Math square Error
def MSE(Y_predict, Y_test, elp):
  total = 0
  for i in range(len(Y_test)):
    total = total + (abs(Y_predict[i]-Y_test[i])-elp)*(abs(Y_predict[i]-Y_test[i])-elp)
  error = total/(len(Y_test))
  return error
```

- Then used 5 fold cross-validation to find the value of y for the test cases and also predict the error.
- The matrices are then initialized as follows:

```
n = len(X_train)
P = P_Matrix_linear(X_train, cons)
q = q_matrix(Y_train, elp)
G = G_matrix(n)
h = h_matrix(C, n)
A = A_matrix(n)
b = b_matrix()
```

- Then converted the matrices in the form of cvxopt matrices:

```
P = cvxopt_matrix(P)
q = cvxopt_matrix(q)
G = cvxopt_matrix(G)
h = cvxopt_matrix(h)
A = cvxopt_matrix(A)
b = cvxopt_matrix(b)
```

- Using the below equations, solved our problem and got our x, which is lamda here, and then split it into lamda_1(a) and lamda_2(a^)

```
sol = cvxopt_solvers.qp(P, q, G, h, A, b)
lamda = np.array(sol['x'])
l = int(len(lamda)/2)
lamda_1 = lamda[0:l, :]
lamda_2 = lamda[l:,:]
```

- Calculated the support vector indices as follows:

```
# the supp_vector array will contain all indics of vectors which are support vectors
supp_vector = []
for j in range(len(lamda_1)):
  temp = lamda_1[j][0]
  if temp>0 and temp<C:
    supp_vector.append(j)
```

- Then calculate the value of b to be used in the prediction of y.

```
# calculation of b
b = 0
for i1 in supp_vector:
  b = b+Y_train[i1]-elp-b_term_linear(X_train[i1], n, lamda_1, lamda_2, X_train, cons)
b = b/(len(supp_vector))
```

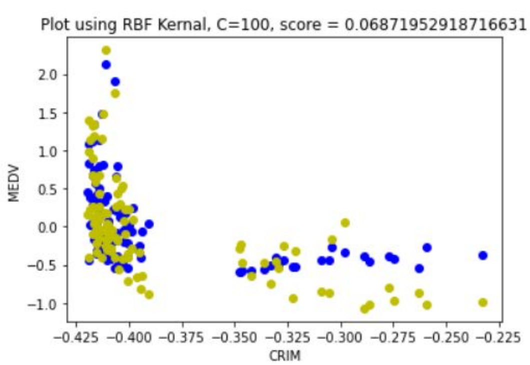
- These codes are used for visualization of data points. Since our dataset is not linear so, I took the **first feature** and plotted the points corresponding to that feature.

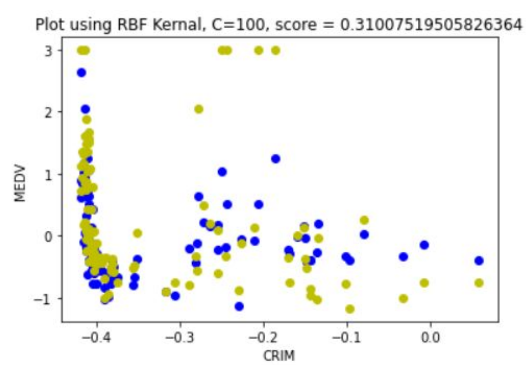
```
# data visualisation
plt.scatter(X_test[:,0], Y_predict_linear, label='Y_predict', color = 'b')
plt.scatter(X_test[:,0], Y_test, label='Y_test', color = 'y')
plt.title("Plot using Linear Kernal, C = 1000, Score = "+str(score_linear[0]))
plt.xlabel("CRIM")
plt.ylabel("MEDV")
plt.show()
```

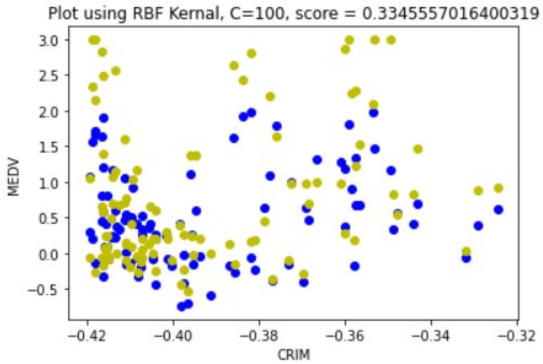
- The error got in :
  SVR using RBF Kernel: 0.298 (Using c = 100, epsilon = 0.1, gamma = 0.1)
  SVR using Linear Kernel: 0.334 (using C = 100)
  SVE using Polynomial Kernal: 17.550 (d = 3, C = 100, epsilon = 0.1)

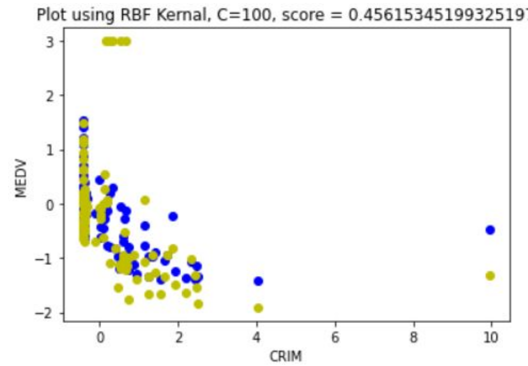
PLOTTING OF GRAPHS

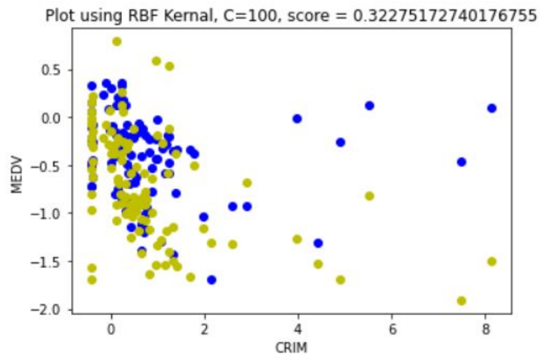
- Plot for a graph using **RBF Kernal** using gamma = 0.1, C = 100, epsilon = 0.1, all 5 figures of cross-validation



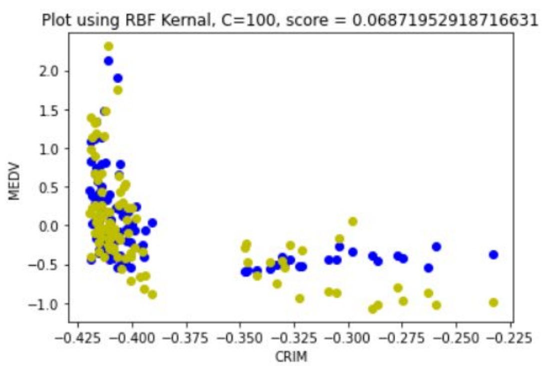


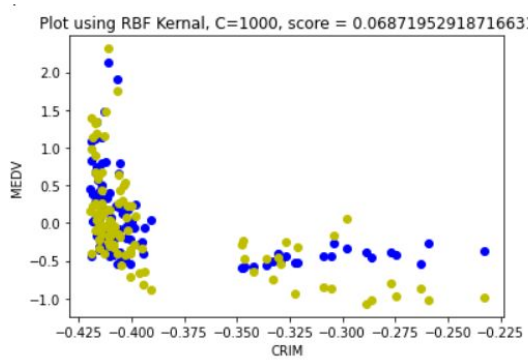


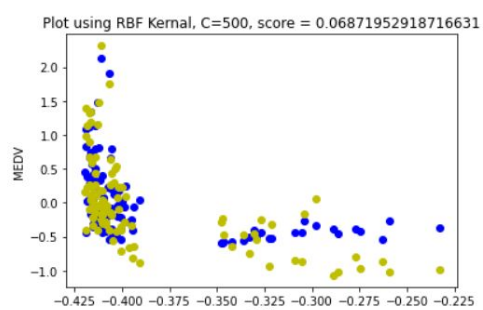




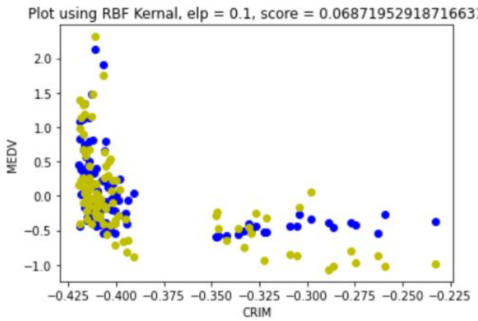
- Plot for different C, keeping epsilon = 0.1, gamma = 0.1

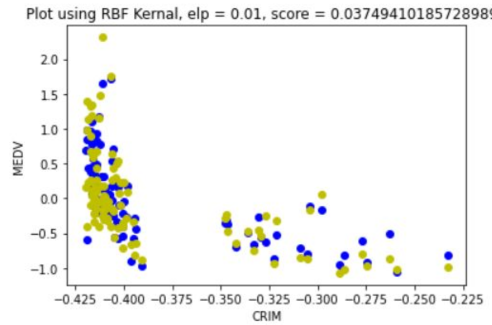


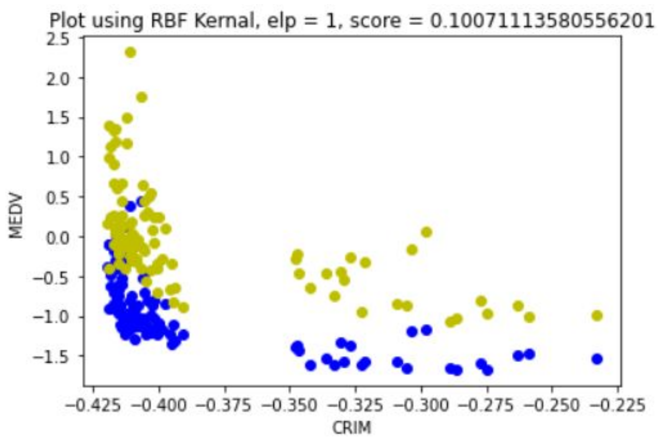




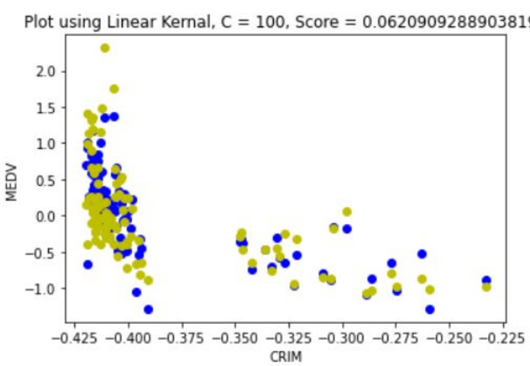
- For different epsilon, keeping gamma = 0.1, C = 100

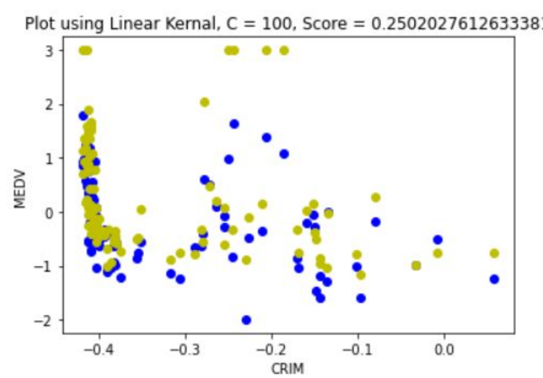


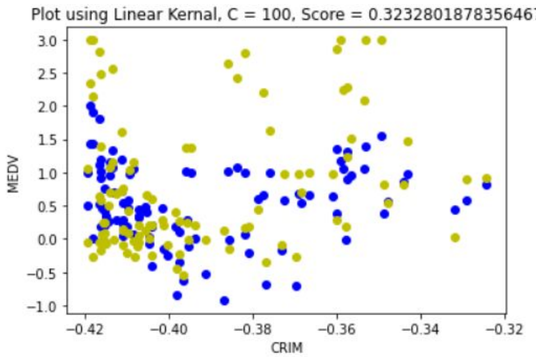


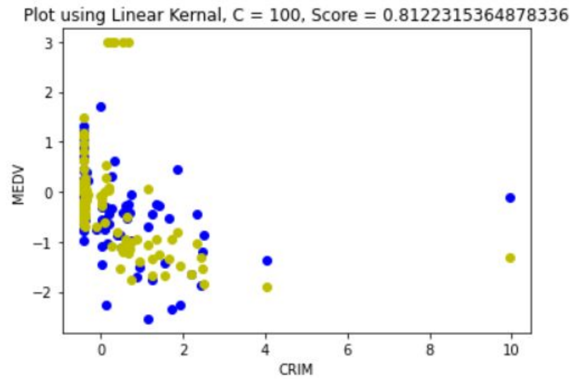


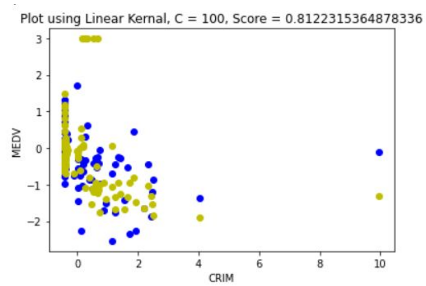
- Plot for a graph using **Linear Kernal** using C = 100



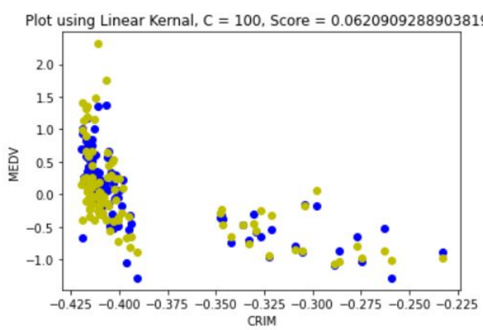


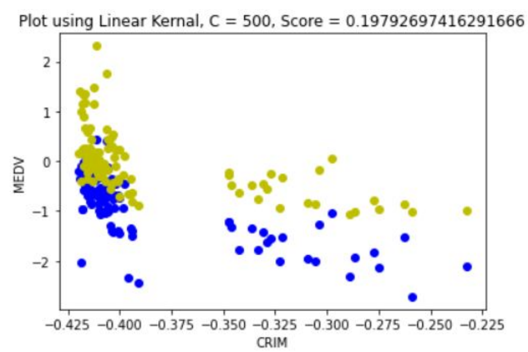


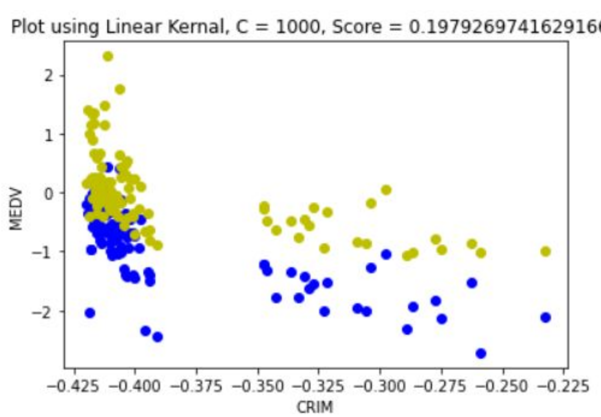




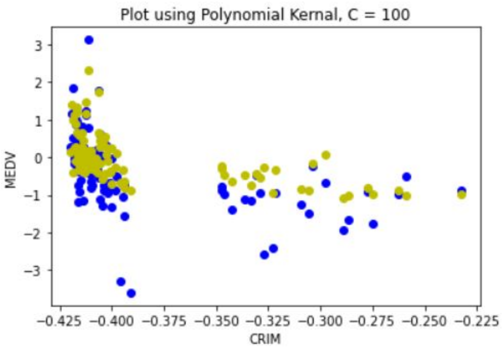
- Plot of graph for different C

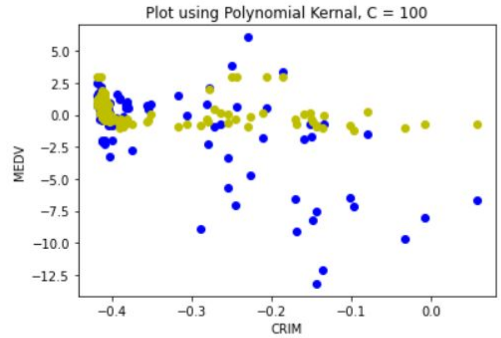


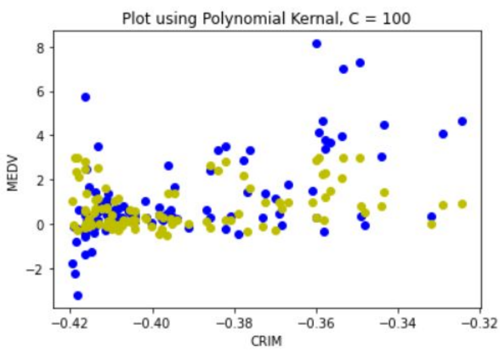


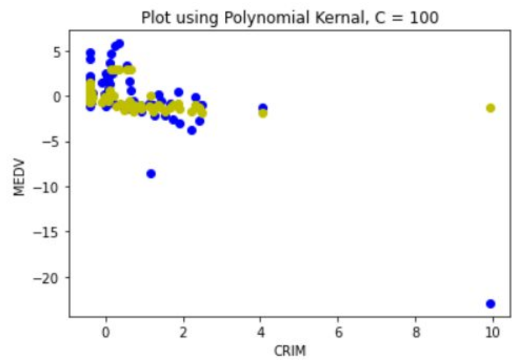


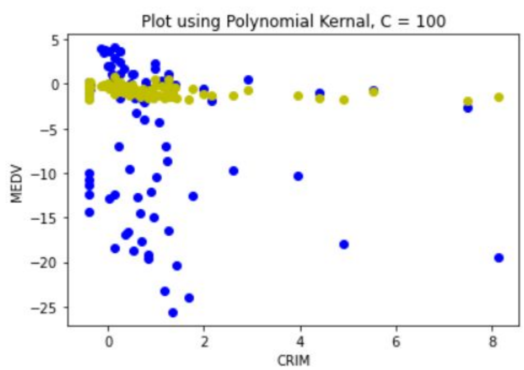
● Plot of Graph for **kernel polynomial** Function



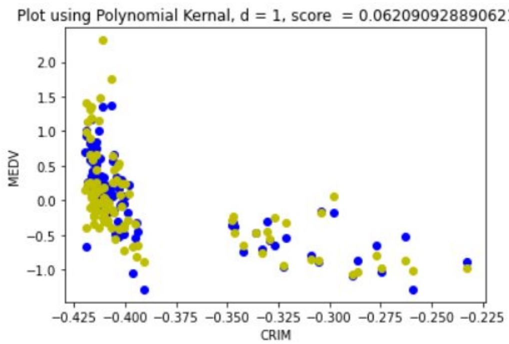


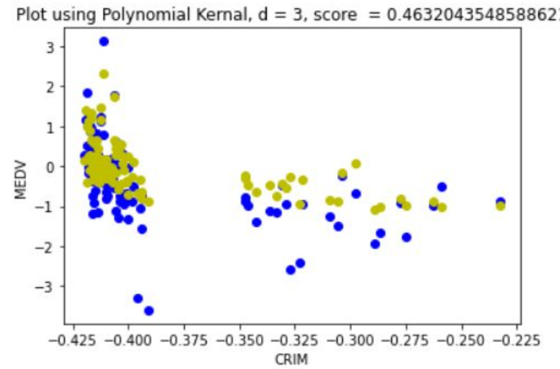


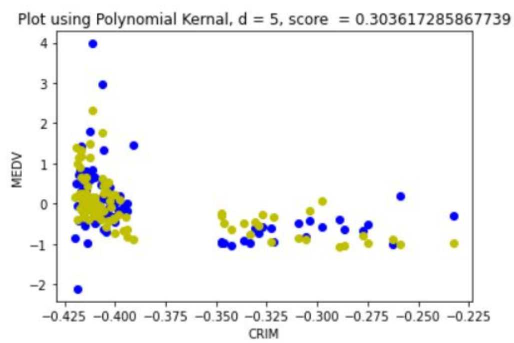


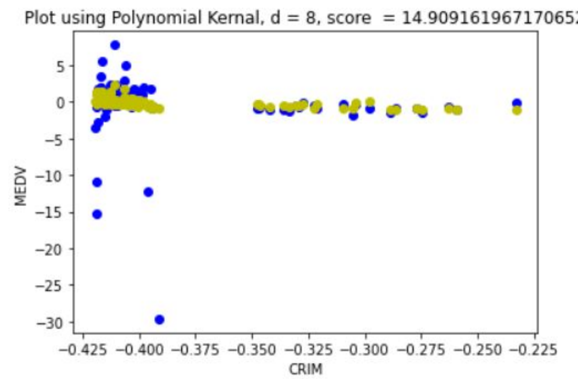


- Polynomial Kernel, for different degree of freedom(d)

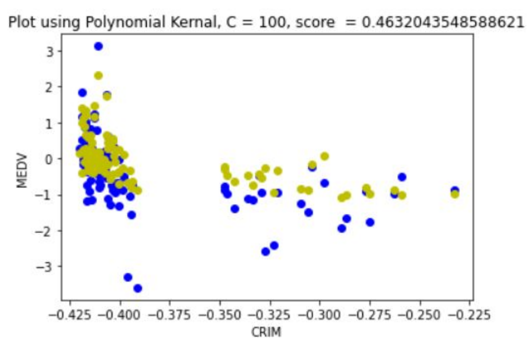


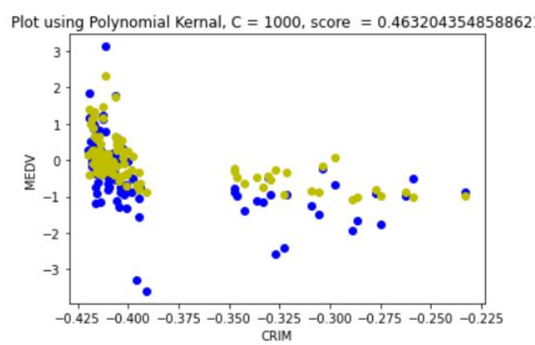


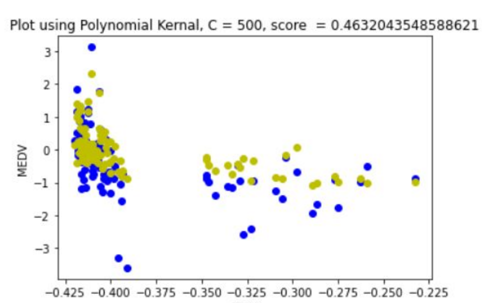




- For different C keeping d = 3







SVR Implementation using sklearn

- Took 5 fold cross-validations.
- The following is the implementation:

```
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=1)
svr_lin = SVR(kernel='linear', C=100, gamma='auto')
svr_poly = SVR(kernel='poly', C=100, gamma='auto', degree=8, epsilon=0.1, coef0=1)
```

- Prediction:

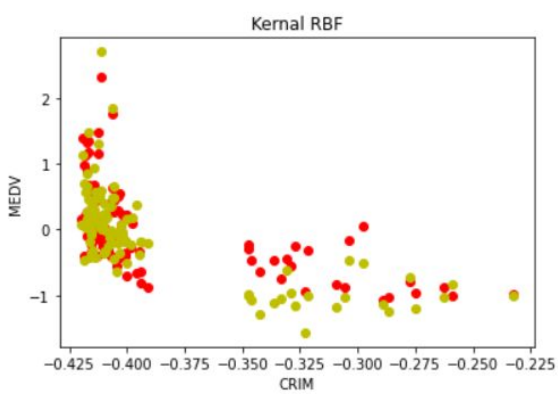
```
svr_poly.fit(X_train, Y_train)
poly_predict = svr_poly.predict(X_test)
scores_poly.append(svr_poly.score(X_test, Y_test))
```

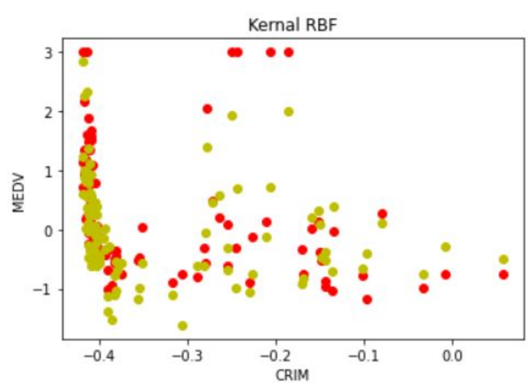
- Codes used for plotting, same plotting as above:

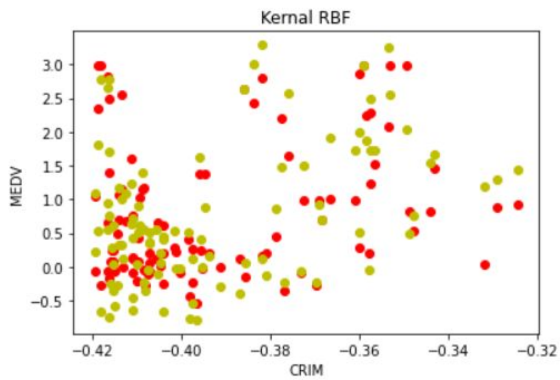
```
plt.title('Kernal Linear, eplison = 0.1, score = '+str(scores_lin[0]))
plt.xlabel('CRIM')
plt.ylabel('MEDV')
plt.scatter(X_test[:,0], Y_test, color = 'g', label="Y_test".format('r'))
plt.scatter(X_test[:,0], lin_predict, color = 'b',label="Y_predicted".format('y'))
plt.show()
```

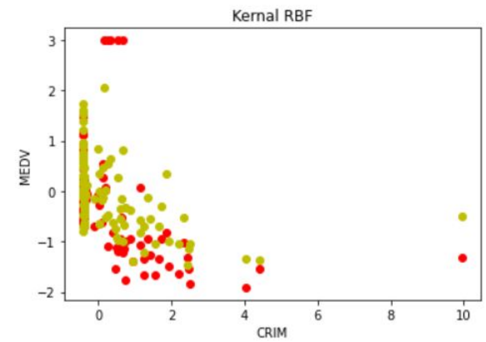
Plotting of Graphs

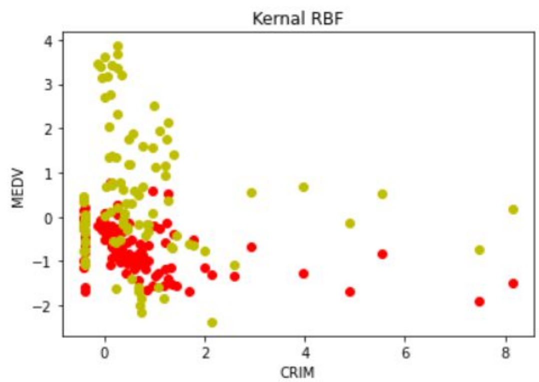
- **RBF Kernal** for C = 100, gamma = 0.1, epsilon = 0.1



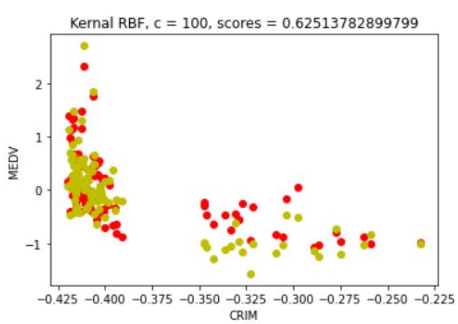


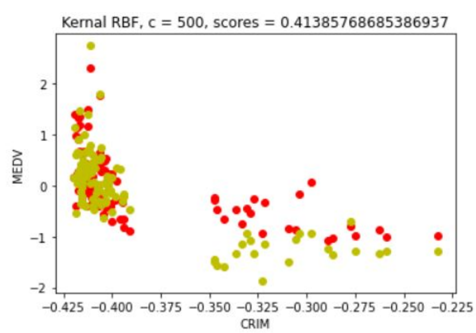


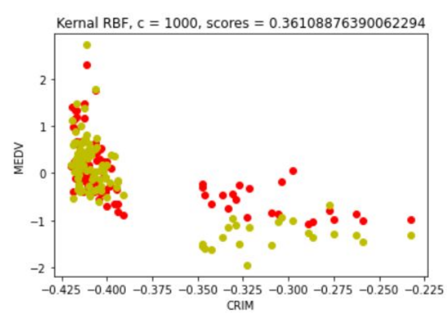




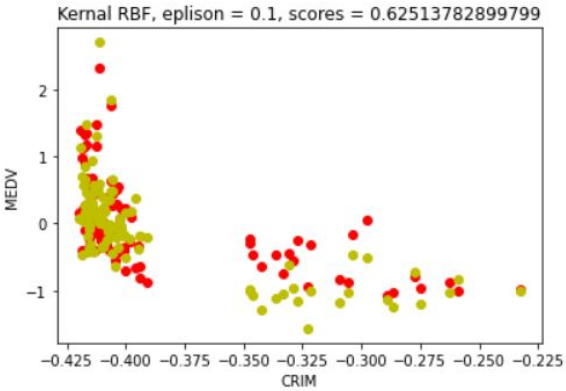
- For different C, keeping other parameters same:

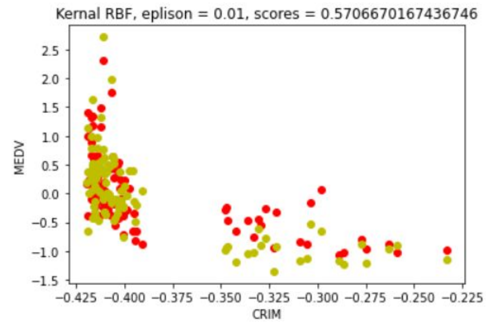


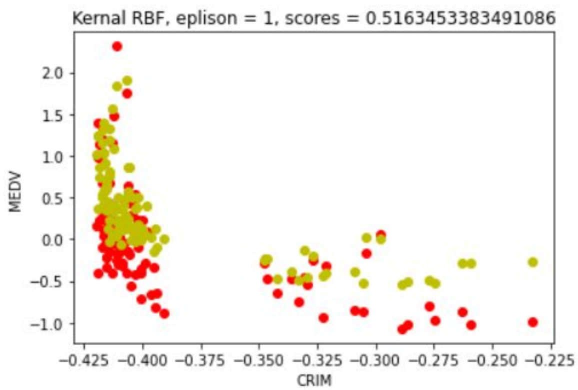




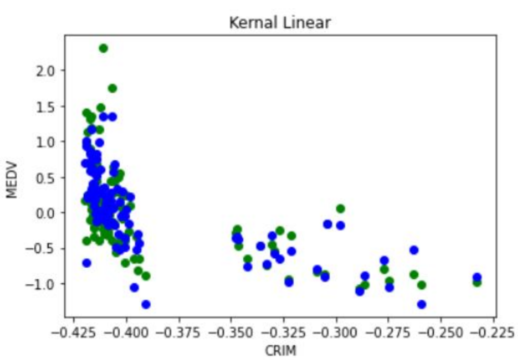
- For different epsilon, keeping C = 100

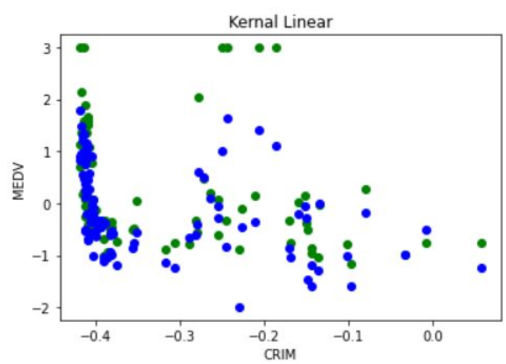


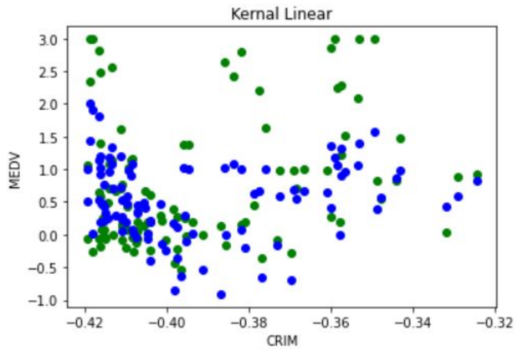


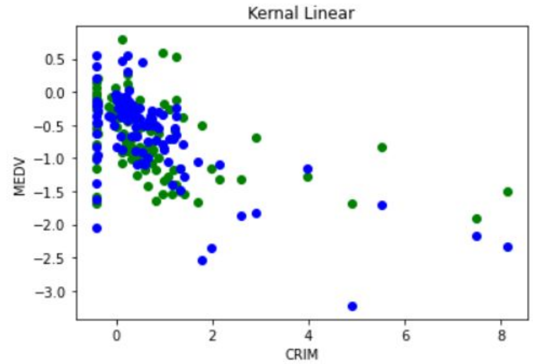


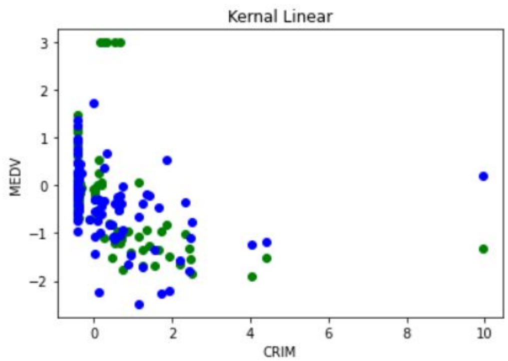
- Plot for **Linear Kernal** for C = 100



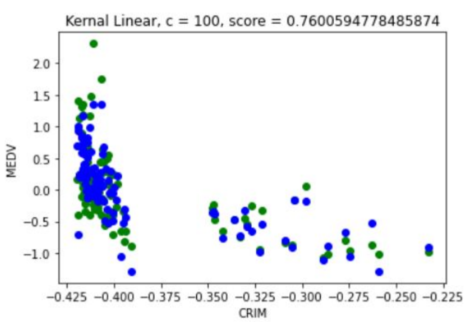


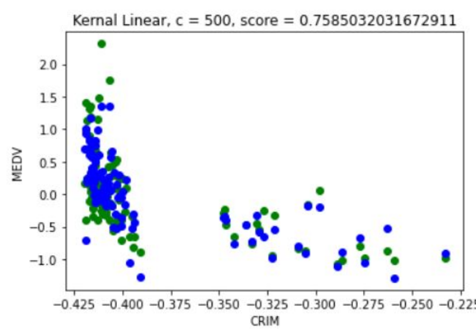


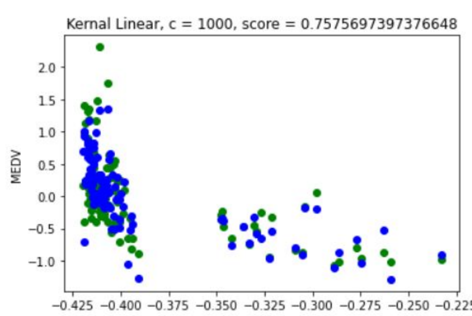




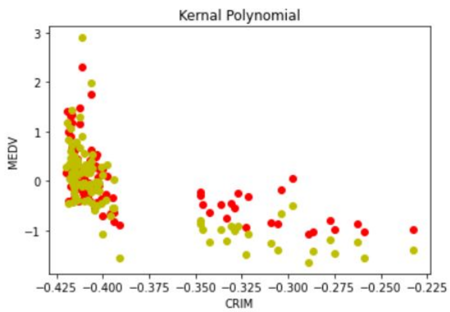
- Plot for Linear Kernel for different C

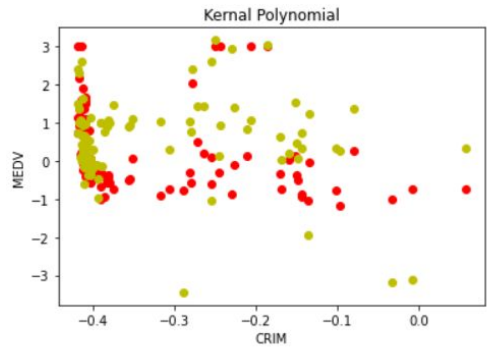


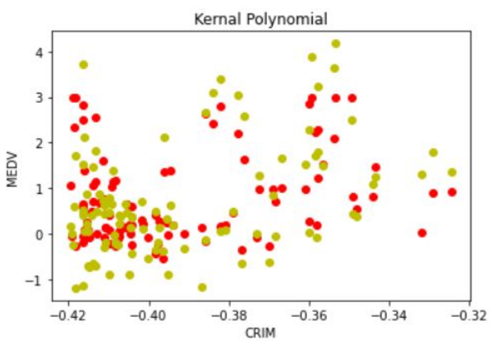


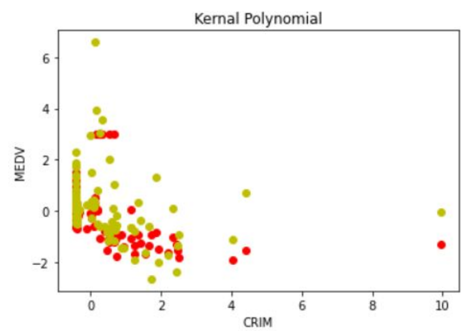


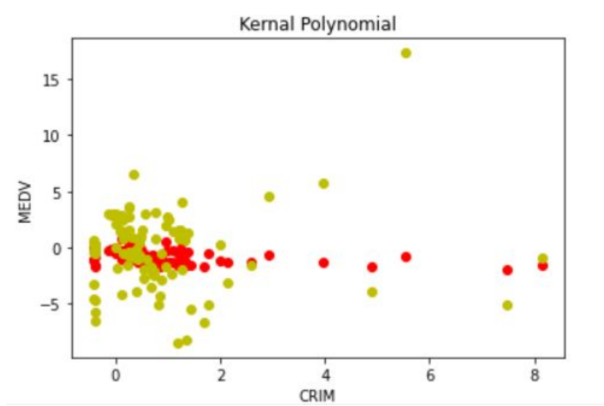
- Plot for **Polynomial Kernal** for C = 100, degree = 3, epsilon = 0.1, coeff0 = 1



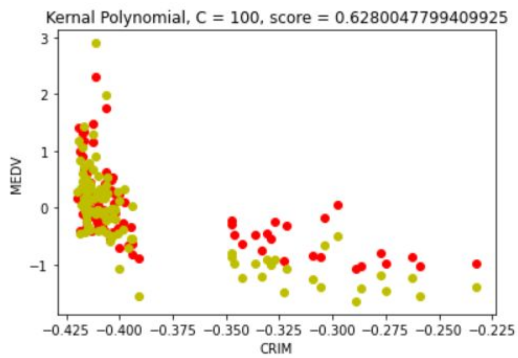


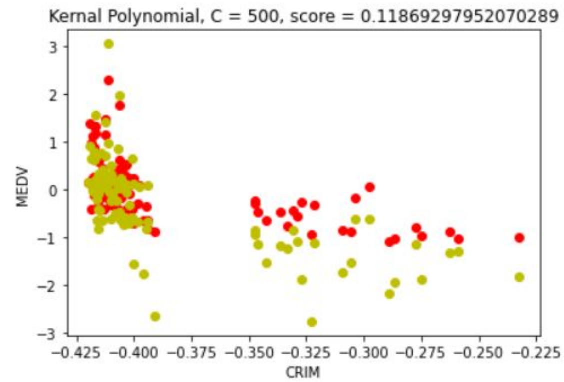


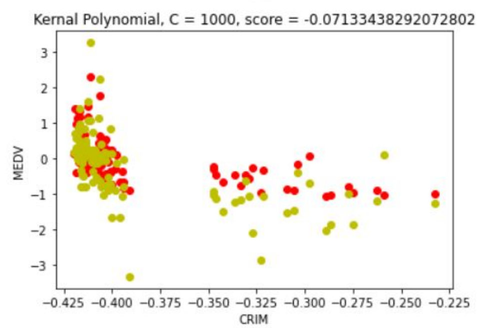




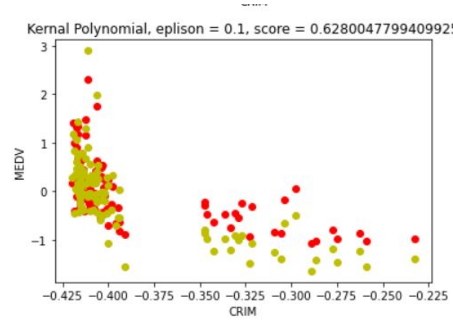
- For different C, keeping other parameters same

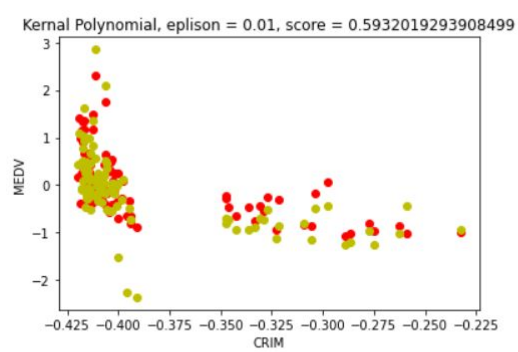


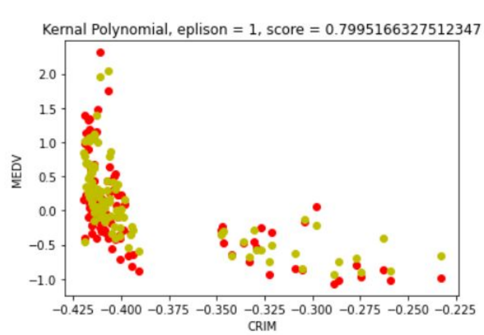


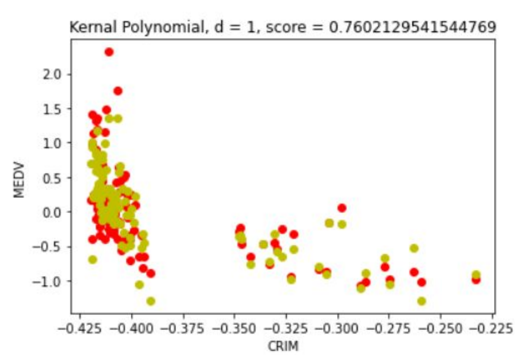


- Plot for different epsilon









- Plot for different degrees of freedom(d)