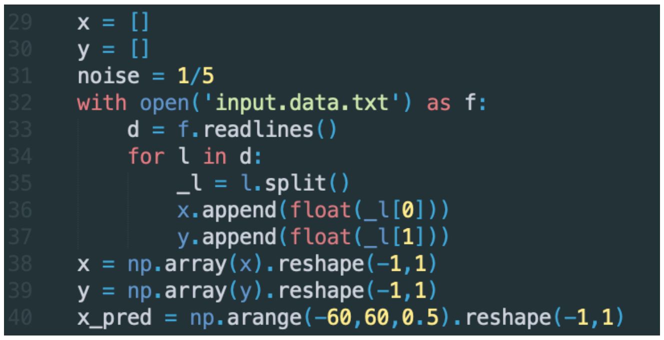
Machine Learning 108-1

HW5

0856733 數據一 黃明翰

Gaussian Process

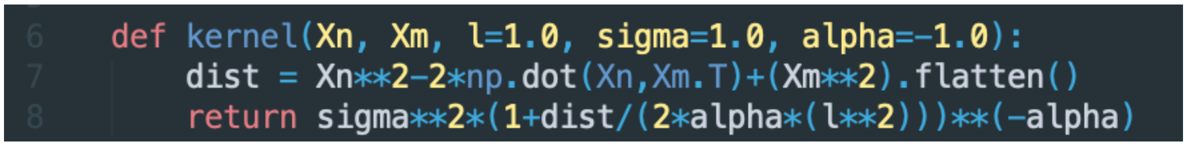
Data



```
29  x = []
30  y = []
31  noise = 1/5
32  with open('input.data.txt') as f:
33      d = f.readlines()
34      for l in d:
35          _l = l.split()
36          x.append(float(_l[0]))
37          y.append(float(_l[1]))
38  x = np.array(x).reshape(-1,1)
39  y = np.array(y).reshape(-1,1)
40  x_pred = np.arange(-60,60,0.5).reshape(-1,1)
```

In Line 29 – Line 40, I read training data (x,y) respectively in ***x*** and ***y***. ***x_pred*** is a list of numbers from -60 to 60 with step 0.5, which I use to predict the distribution of *f* using Gaussian Process Regression.

Problem 1

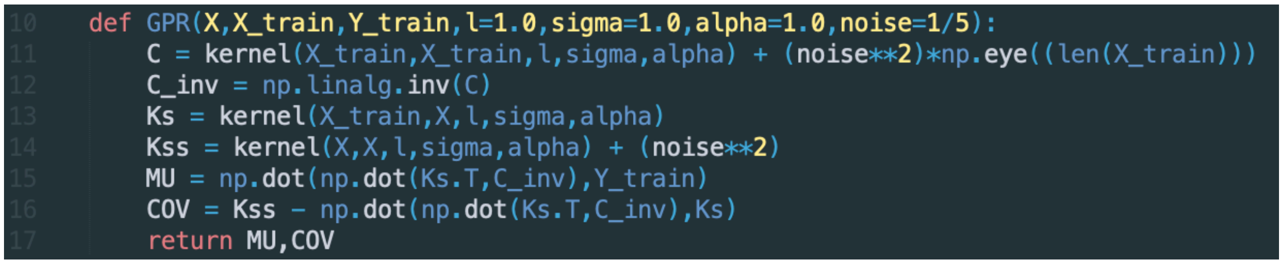


```
6  def kernel(Xn, Xm, l=1.0, sigma=1.0, alpha=-1.0):
7      dist = Xn**2-2*np.dot(Xn,Xm.T)+(Xm**2).flatten()
8      return sigma**2*(1+dist/(2*alpha*(l**2)))**(-alpha)
```

In Line 6 – Line 8, I define the **rational quadratic kernel** which is equivalent to

$$k(x,x') = \sigma^2(1+\frac{(x-x')^2}{2\alpha l^2})^{-\alpha}$$

I will show how these parameters affect the result of Gaussian Process Regression in the last section of this part.



```
10  def GPR(X,X_train,Y_train,l=1.0,sigma=1.0,alpha=1.0,noise=1/5):
11      C = kernel(X_train,X_train,l,sigma,alpha) + (noise**2)*np.eye((len(X_train)))
12      C_inv = np.linalg.inv(C)
13      Ks = kernel(X_train,X,l,sigma,alpha)
14      Kss = kernel(X,X,l,sigma,alpha) + (noise**2)
15      MU = np.dot(np.dot(Ks.T,C_inv),Y_train)
16      COV = Kss - np.dot(np.dot(Ks.T,C_inv),Ks)
17      return MU,COV
```

In Line 10 – Line 17, I define the function of Gaussian Process Regression, which takes in ***X*** for prediction, ***X_train*** and ***Y_train*** for training data, ***hyperparameters*** and ***noise*** $\boldsymbol{\beta}$.

C is the covariance matrix of training part which has elements:

$$C(x_n, x_m) = k(x_n, x_m) + \beta^{-1}\delta_{nm}$$

Then, for the prediction part, we have:

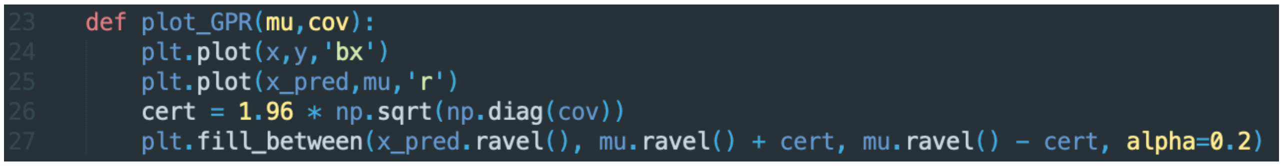
$$C_{N+1} = \begin{bmatrix} C & k(x, x^*) \\ k(x, x^*)^T & k(x^*, x^*) + \beta^{-1} \end{bmatrix}$$

And the conditional distribution $p(y^*|y)$ is a Gaussian distribution with:

$$\begin{aligned} &\mu(x) = k(x, x^*)^T C^{-1} y \\ &\sigma^2(x) = \ k^* - \ k(x, x^*)^T C^{-1} k(x, x^*) \\ &where\ k^* = \ k(x^*, x^*) + \beta^{-1} \end{aligned}$$

Here, my ***Ks*** stands for $k(x, x^*)$ and ***Kss*** stands for k^*.

This function will return $\mu(x)$ and $\sigma^2(x)$, which I can use to draw the plot of a line to represent mean of *f* and the 95% confidence interval of *f*. Following is code for visualizing and the result (I set all hyperparameters to 1 for default temporary).



```
23  def plot_GPR(mu,cov):
24      plt.plot(x,y,'bx')
25      plt.plot(x_pred,mu,'r')
26      cert = 1.96 * np.sqrt(np.diag(cov))
27      plt.fill_between(x_pred.ravel(), mu.ravel() + cert, mu.ravel() - cert, alpha=0.2)
```

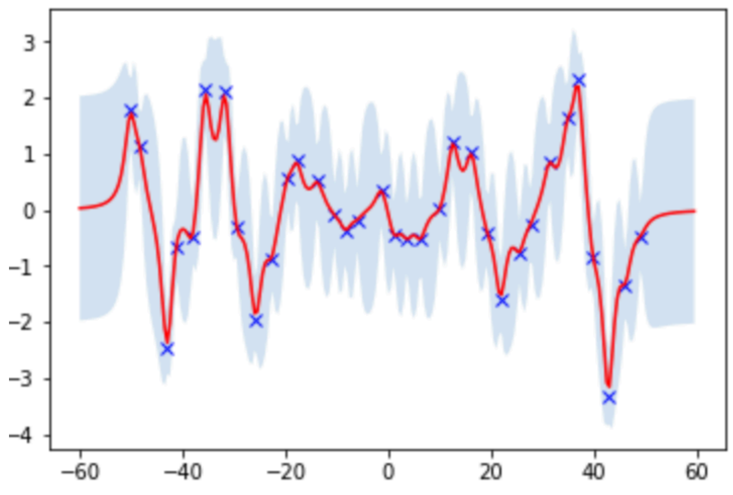


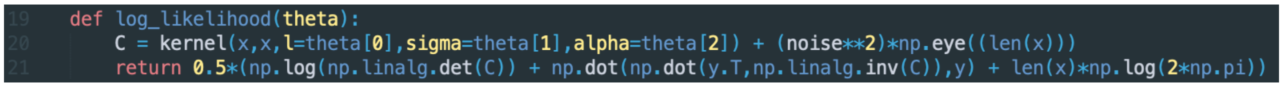
fig. 1 Result of GPR

Problem 2

The negative log marginal likelihood we want to minimize:

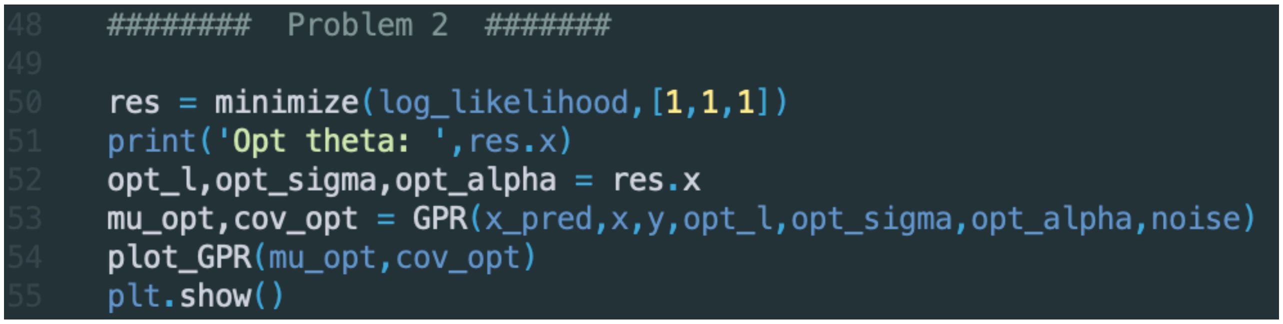
$$-\ln p(y|\theta) = \frac{1}{2}\ln|C_\theta| + \frac{1}{2}y^T C_\theta^{-1} y + \frac{N}{2}\ln(2\pi)$$

Which is defined at the Line 19 – Line 21:



```
19  def log_likelihood(theta):
20      C = kernel(x,x,l=theta[0],sigma=theta[1],alpha=theta[2]) + (noise**2)*np.eye((len(x)))
21      return 0.5*(np.log(np.linalg.det(C)) + np.dot(np.dot(y.T,np.linalg.inv(C)),y) + len(x)*np.log(2*np.pi))
```

In Line 48 – Line 55, I use the minimize function from scipy.optimize to find the optimal hyperparameters for the training data and then use them to predict the distribution of *f*.



```
48  ########  Problem 2  #######
49
50  res = minimize(log_likelihood,[1,1,1])
51  print('Opt theta: ',res.x)
52  opt_l,opt_sigma,opt_alpha = res.x
53  mu_opt,cov_opt = GPR(x_pred,x,y,opt_l,opt_sigma,opt_alpha,noise)
54  plot_GPR(mu_opt,cov_opt)
55  plt.show()
```

The following figure is the result of Gaussian Process Regression with optimal hyperparameters:

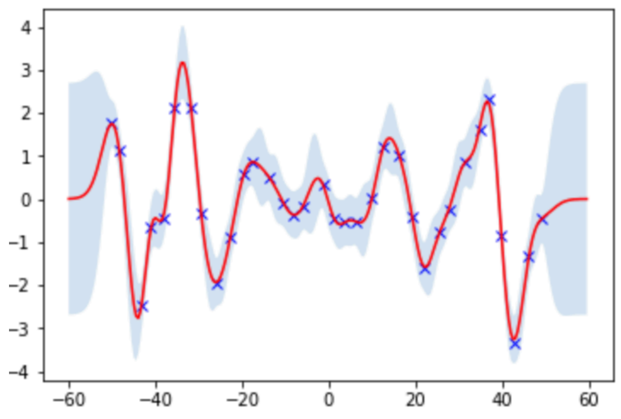
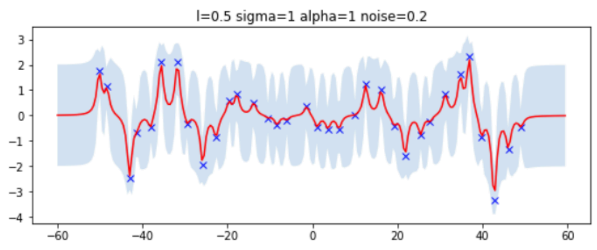


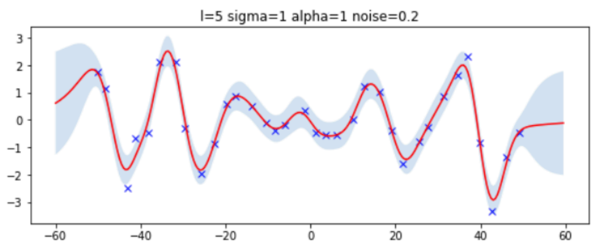
fig. 2 Result of Gaussian Process Regression with optimal hyperparameters

From the result plot, we can see that the result is way better than the result of using default hyperparameters in problem 1. The function is smoother and has lower confidence interval.

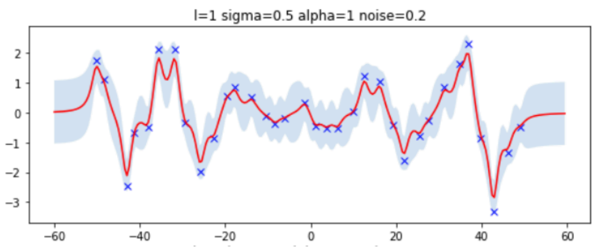
Affect from the hyperparameters

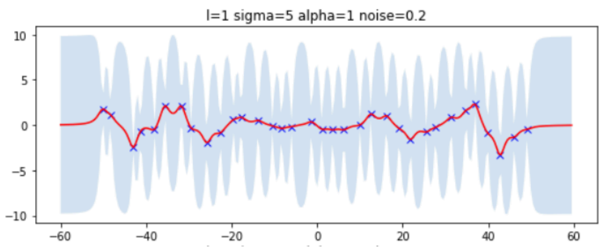
In Line 59 – Line 75, I try different sets of hyperparameters to observe how these hyperparameters affect our result.



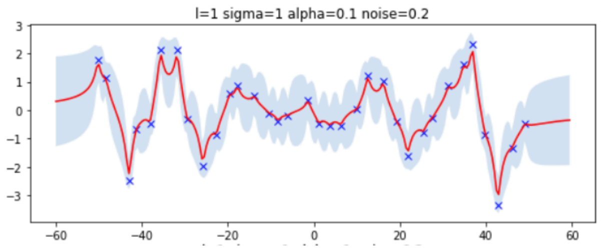


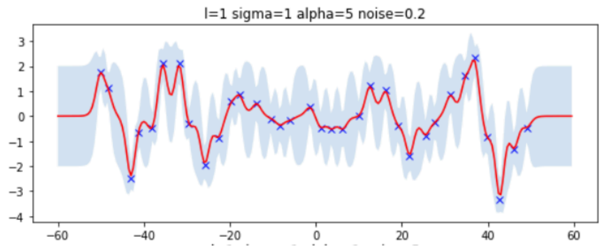
First, we can see the higher l value lead to smoother function with narrow confidence interval and coarser approximation to the training data.



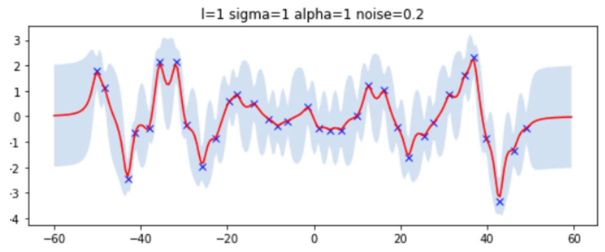


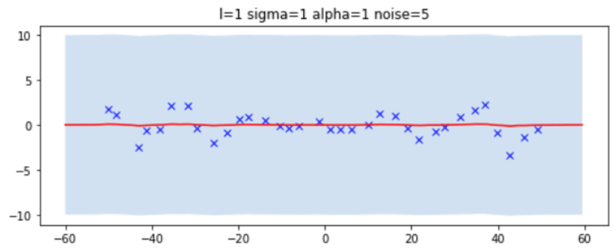
Secondly, the σ value controls the vertical variation of the function.





The higher α gives wider and sharper confidence interval and smoother function. Notice that when $\alpha \rightarrow \infty$, this kernel is equal to RBF kernel.



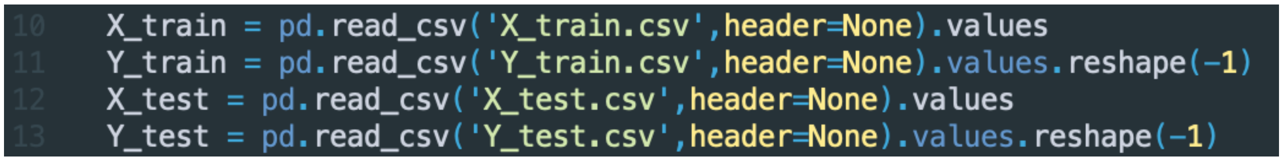


The higher *noise* value results in coarser approximation to the training data, which avoids overfitting to noisy data.

SVM on MNIST dataset

The **OUTPUT** of console is in **SVM_OUT.TXT** file.

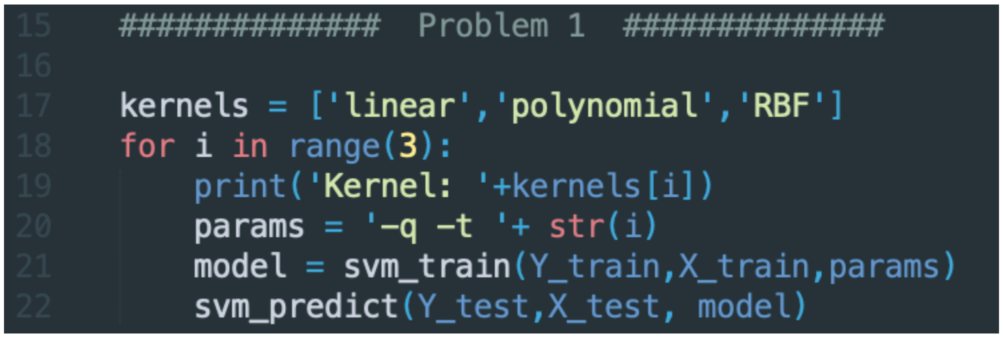
Data



```
10    X_train = pd.read_csv('X_train.csv',header=None).values
11    Y_train = pd.read_csv('Y_train.csv',header=None).values.reshape(-1)
12    X_test = pd.read_csv('X_test.csv',header=None).values
13    Y_test = pd.read_csv('Y_test.csv',header=None).values.reshape(-1)
```

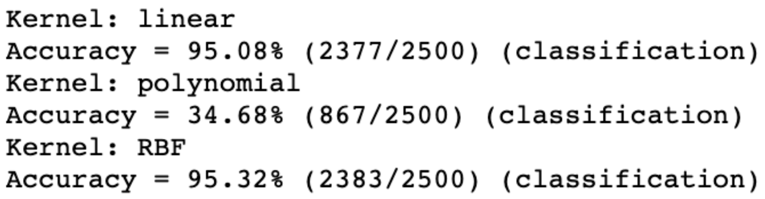
In Line 10 – Line 13, I read training data and testing data into ***X_train, Y_train, X_test*** and ***Y_test*** respectively.

Problem 1



```
15    ##############  Problem 1  ##############
16
17    kernels = ['linear','polynomial','RBF']
18    for i in range(3):
19        print('Kernel: '+kernels[i])
20        params = '-q -t '+ str(i)
21        model = svm_train(Y_train,X_train,params)
22        svm_predict(Y_test,X_test, model)
```

In Line 15 – Line 22, I try three kernels (linear, polynomial and RBF kernel) respectively. I train the model with training data in Line 21 and use this model to classify testing images. The results are:



```
Kernel: linear
Accuracy = 95.08% (2377/2500) (classification)
Kernel: polynomial
Accuracy = 34.68% (867/2500) (classification)
Kernel: RBF
Accuracy = 95.32% (2383/2500) (classification)
```

fig. 3 Results of three kernels with default setting

From the result, we can see that the RBF kernel has the best performance with 95.32% accuracy and the polynomial kernel only performance 34.68% accuracy, which is not very good.

Problem 2

In problem 2, in order to find best parameters, we use grid search for finding parameters of best performing model. In Line 24 – Line 65 I set the **number of folds of cross-validation to 5**, and try parameter $\boldsymbol{C}$ from $\log_2 C = -5$ to $\log_2 C = 5$ for all three kernels.

For linear kernel, $\boldsymbol{C}$ is the only parameter we need to tune.
For polynomial kernel, except of $\boldsymbol{C}$, we have $\boldsymbol{g}$ for γ, $\boldsymbol{d}$ for degree and $\boldsymbol{r}$ for coef0 to tune. I try parameter $\boldsymbol{g}$ from $\log_2 g = -5$ to $\log_2 g = 5$, $\boldsymbol{d}$ from 2 to 4 and $\boldsymbol{r}$ from 0 to 2
For RBF kernel, we have two parameters to tune: C and $\boldsymbol{g}$ for γ. I try parameter $\boldsymbol{g}$ from $\log_2 g = -5$ to $\log_2 g = 5$.

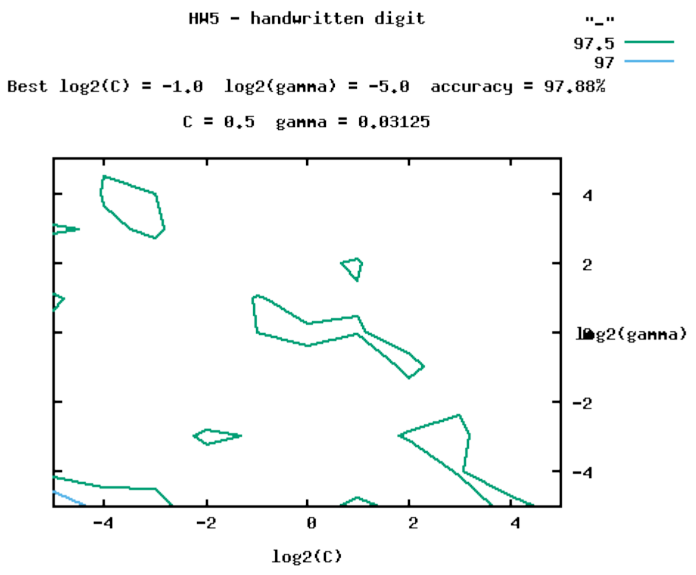
After searching, we get the best parameters and best performance of CV respectively:
[Linear] c = 0.03125, acc=97.16%
[Polynomial] c=0.5, g=0.03125, d=3, r=1, acc=98.34%
[RBF] c=0.5 g=0.03125, acc=97.88%

Then, in Line 103 – Line 116, I try these best performing models on testing data and get the following accuracy:

Linear: 96%, Polynomial: 97.92%, RBF: 98.04%

All of them are better than the result of problem1.

Also, in Line 67 – Line 101, I use *gnuplot* to draw the result of the grid search of RBF kernel, which only has two parameters that is suitable to use a 2D plot to visualize:



Problem 3

The linear kernel has the form:

$$k(x, x') = x^T x'$$

The RBF kernel has the form:

$$k(x, x') = exp(-\gamma \|x - x'\|_2^2)$$

In Line 119 – Line 130, I set the gamma for RBF kernel equal to 0.03125, which is the best gamma from previous problem. To use linear+RBF kernel, I sum the kernels up, and in order to use the precompute kernel in Libsvm, we have to add one more column of index numbers as the first column of the precompute kernel matrix.
The *cdist*, *pdist* and *squareform* function here is from *scipy.spatial.distance*. Function *cdist* and *pdist* here compute the squared Euclidean distance $\|u - v\|_2^2$ between the vectors. The *squareform* function here is to help matrix addition.

The result of my linear+RBF kernel perform **95.32%** accuracy.

Notes

The details of parameter setting of libsvm is here:

```
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC
        1 -- nu-SVC
        2 -- one-class SVM
        3 -- epsilon-SVR
        4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/k)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 40)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates: whether to train an SVC or SVR model for probability estimates, 0 or 1 (default 0)
-wi weight: set the parameter C of class i to weight*C in C-SVC (default 1)
-v n: n-fold cross validation mode
```