**Machine Learning Homework 5**

**Gaussian Process & SVM**

**數據所碩一葉詠富310554031**

**Github:** [**https://github.com/frankye1000/NYCU-MachineLearning/tree/master/HW5**](https://github.com/frankye1000/NYCU-MachineLearning/tree/master/HW5)

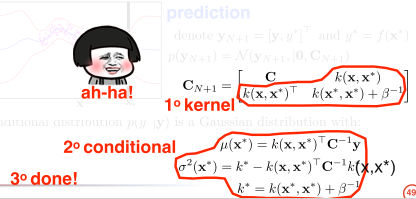
**1. Gaussian Process**

**Part1:** Apply Gaussian Process Regression to predict the distribution of f and visualize the result.

First thing is to load data, I define a load data function.

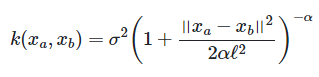


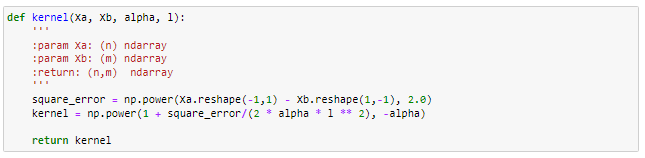
Use the **three steps** mentioned by the teacher in class.

****

Step1. Rational quadratic kernel

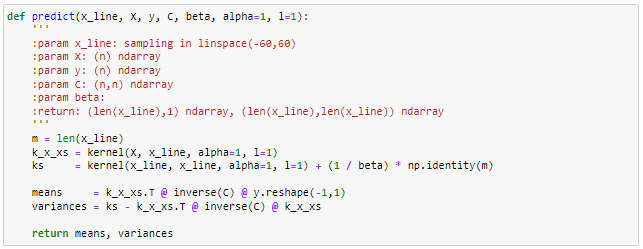
Define the rational quadratic kernel by below formula.

****



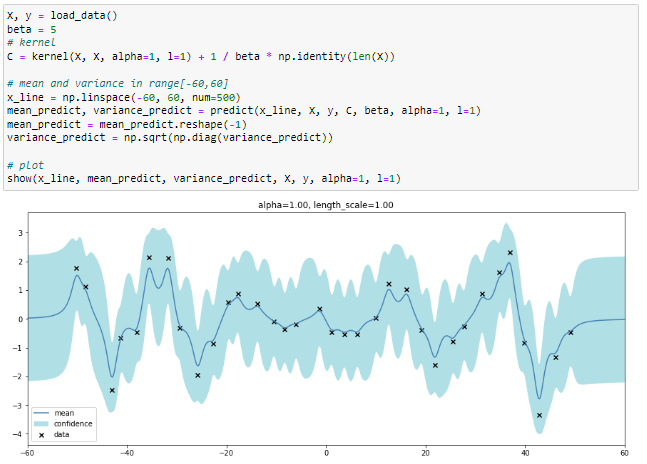
Step2. Conditional

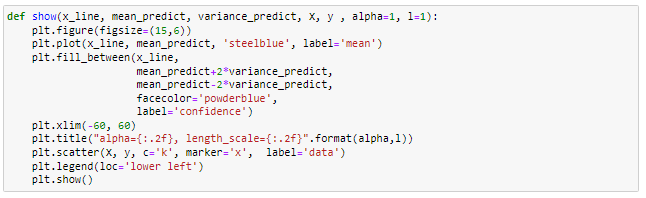
Use teacher formula to calculate mean & variance.



Step3. Done!

I set initial parameter beta=5, alpha=1, length scale=1, x\_line=[-60,60], and define show function to plot the predict result.





**Part2:** Optimize the kernel parameters by minimizing negative marginal log-likelihood, and visualize the result again.

Find alpha & length scale when minimum log likelihood by below formula.

D:\NYCU\NYCU-MachineLearning\HW5\img\optimize_loglikelihood_new.PNG

I define a log likelihood function, and use scipy.optimize.minimize to find minimum alpha & length scale.



I set the initial value alpha & length scale = [0.01, 0.1, 0, 10, 100] and set the bound of alpha & length scale [10^-5, 10^5] to find the minimum value.

If the result is compared with alpha=1, length scale=1, the variance of each point becomes smaller.

**I find the optimize (alpha, length scale)=(2661, 2.96).**



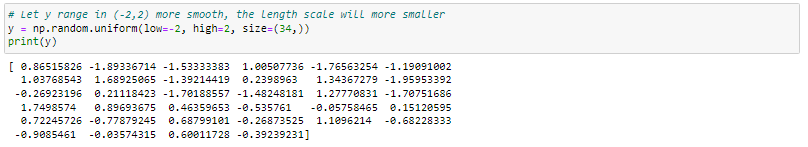
**Observation**

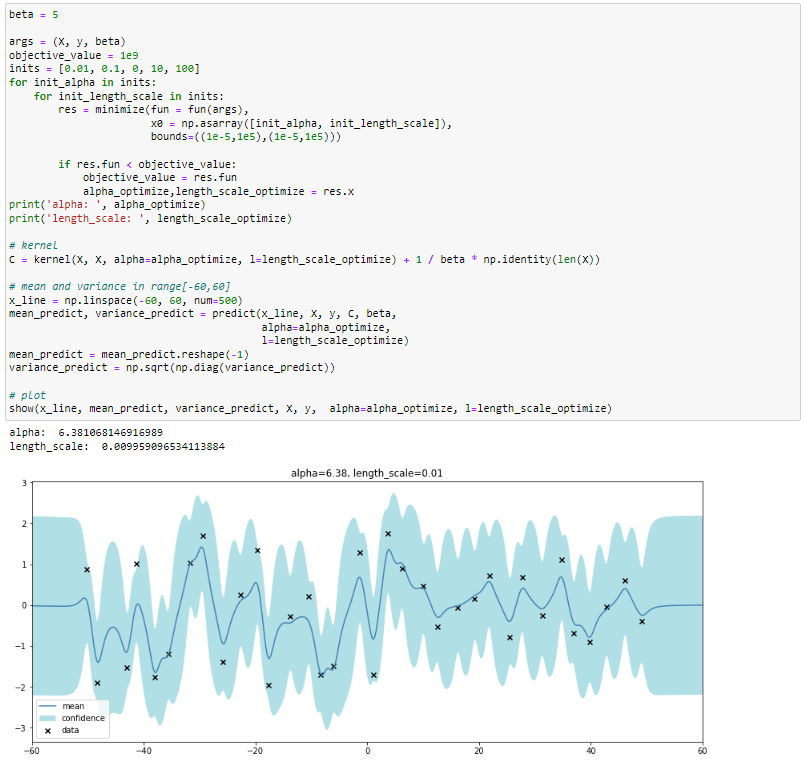
As you increase the alpha & length scale, the learn functions keep getting smoother.

I do a test to make the value of y range (-2,2), which is more smoother.

Then the best alpha & length scale become smaller.

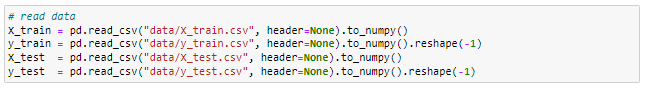
**Optimize (alpha, length scale)=(6.38, 0.0099).**





**2. SVM**

First thing is to load data. I use pandas to load csv data.



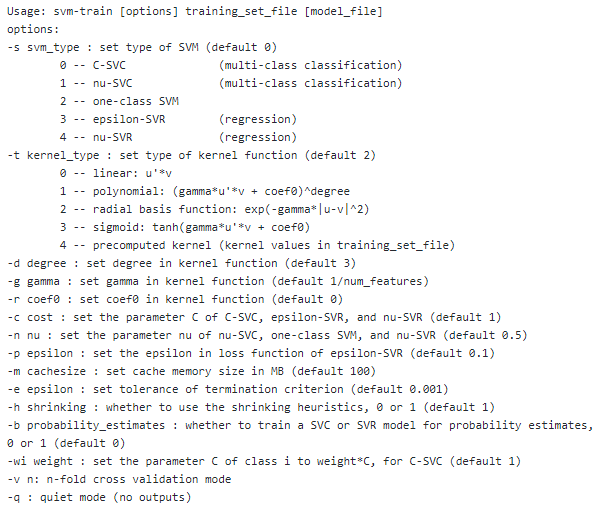
**Part1:** Use different kernel functions (linear, polynomial, and RBF kernels) and have comparison between their performance.

Use the package libsvm, have same important parameter.

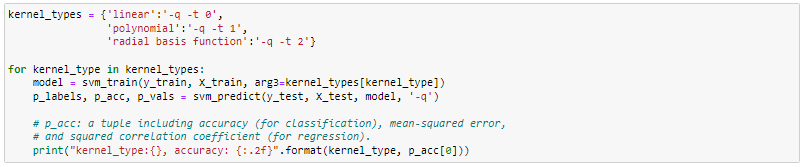
-t means the different kernel type,

-q means will not return calculation process.

reference from <https://github.com/cjlin1/libsvm>



I change kernel type and use default parameter to predict X\_test and compare with ground truth.



|  |  |
| --- | --- |
| Kernel type | accuracy |
| Linear | 95.08% |
| Polynomial | 34.68% |
| Radial basis function | 95.32% |

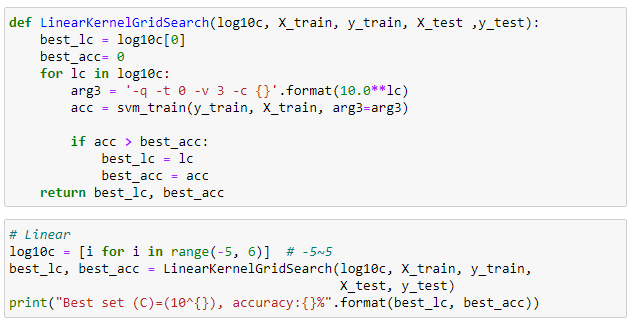
**Part2:** Please use C-SVC. please do the grid search for finding parameters of the best performing model. For instance, in C-SVC you have a parameter C, and if you use RBF kernel you have another parameter 𝛾, you can search for a set of (C, 𝛾) which gives you best performance in cross-validation.

2021/12/02 TA:In part 2, please do grid search also for the kernel functions mentioned in part 1 and find the best parameters for **each** kernel

I define **LinearKernelGridSearch** function to search the best combination of **C and accuracy**.

The cross validation = 3 to get the average accuracy.

The C= 10^-5~10^ 5, and get the best set C= ,accuracy=97.18%.



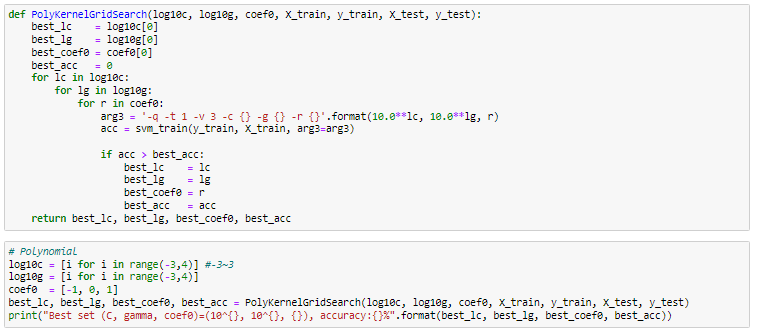
|  |  |
| --- | --- |
|  | Accuracy |
| -5 | 79.32% |
| -4 | 88.32% |
| -3 | 95.28% |
| -2 | 96.92% |
| -1 | **97.18%** |
| 0 | 96.36% |
| 1 | 96.02% |
| 2 | 96.20% |
| 3 | 96.02% |
| 4 | 95.96% |
| 5 | 96.16% |

I define a **PolyKernelGridSearch** function to search the best combination of **(C, 𝛾, coed0) and accuracy.**

The cross validation = 3 to get the average accuracy.

The C= 10^-3~10^ 3, gamma=10^-3~10^ 3, coef0= [-1,0,1],

and get the best set (C, 𝛾,coef0)=(,, 1), accuracy=98.04%.

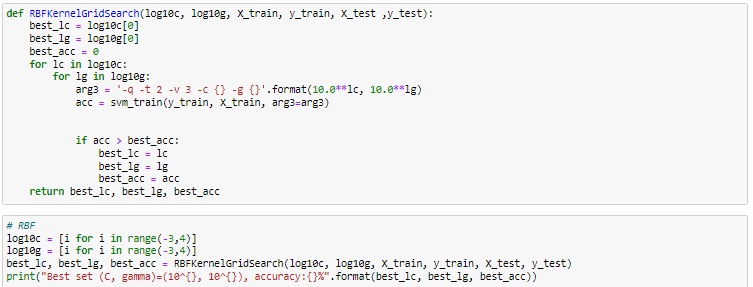


|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | coef0 | Accuracy |  |  | coef0 | Accuracy |  |  | coef0 | Accuracy |
| -3 | -3 | -1 | 82.12% | -1 | -1 | 0 | 97.40% | 1 | 1 | 1 | 97.62% |
| -3 | -3 | 0 | 28.78% | -1 | -1 | 1 | 97.96% | 1 | 2 | -1 | 97.50% |
| -3 | -3 | 1 | 77.06% | -1 | 0 | -1 | 97.18% | 1 | 2 | 0 | 97.42% |
| -3 | -2 | -1 | 84.70% | -1 | 0 | 0 | 97.50% | 1 | 2 | 1 | 97.48% |
| -3 | -2 | 0 | 28.48% | -1 | 0 | 1 | 97.44% | 1 | 3 | -1 | 97.48% |
| -3 | -2 | 1 | 76.02% | -1 | 1 | -1 | 97.54% | 1 | 3 | 0 | 97.26% |
| -3 | -1 | -1 | 92.28% | -1 | 1 | 0 | 97.44% | 1 | 3 | 1 | 97.50% |
| -3 | -1 | 0 | 96.12% | -1 | 1 | 1 | 97.40% | 2 | -3 | -1 | 95.54% |
| -3 | -1 | 1 | 97.42% | -1 | 2 | -1 | 97.52% | 2 | -3 | 0 | 89.24% |
| -3 | 0 | -1 | 97.38% | -1 | 2 | 0 | 97.34% | 2 | -3 | 1 | 96.94% |
| -3 | 0 | 0 | 97.60% | -1 | 2 | 1 | 97.38% | 2 | -2 | -1 | 76.92% |
| -3 | 0 | 1 | 97.30% | -1 | 3 | -1 | 97.36% | 2 | -2 | 0 | 97.48% |
| -3 | 1 | -1 | 97.36% | -1 | 3 | 0 | 97.54% | 2 | -2 | 1 | 97.70% |
| -3 | 1 | 0 | 97.64% | -1 | 3 | 1 | 97.26% | 2 | -1 | -1 | 95.08% |
| -3 | 1 | 1 | 97.50% | 0 | -3 | -1 | 96.08% | 2 | -1 | 0 | 97.44% |
| -3 | 2 | -1 | 97.22% | 0 | -3 | 0 | 28.58% | 2 | -1 | 1 | 97.96% |
| -3 | 2 | 0 | 97.34% | 0 | -3 | 1 | 96.32% | 2 | 0 | -1 | 97.20% |
| -3 | 2 | 1 | 97.22% | 0 | -2 | -1 | 79.84% | 2 | 0 | 0 | 97.44% |
| -3 | 3 | -1 | 97.36% | 0 | -2 | 0 | 96.20% | 2 | 0 | 1 | 97.60% |
| -3 | 3 | 0 | 97.48% | 0 | -2 | 1 | 97.74% | 2 | 1 | -1 | 97.48% |
| -3 | 3 | 1 | 97.38% | 0 | -1 | -1 | 94.98% | 2 | 1 | 0 | 97.24% |
| -2 | -3 | -1 | 81.76% | 0 | -1 | 0 | 97.42% | 2 | 1 | 1 | 97.50% |
| -2 | -3 | 0 | 28.72% | **0** | **-1** | **1** | **98.04%** | 2 | 2 | -1 | 97.78% |
| -2 | -3 | 1 | 77.42% | 0 | 0 | -1 | 97.32% | 2 | 2 | 0 | 97.54% |
| -2 | -2 | -1 | 89.38% | 0 | 0 | 0 | 97.66% | 2 | 2 | 1 | 97.08% |
| -2 | -2 | 0 | 58.80% | 0 | 0 | 1 | 97.44% | 2 | 3 | -1 | 97.40% |
| -2 | -2 | 1 | 94.46% | 0 | 1 | -1 | 97.36% | 2 | 3 | 0 | 97.48% |
| -2 | -1 | -1 | 95.74% | 0 | 1 | 0 | 97.20% | 2 | 3 | 1 | 97.24% |
| -2 | -1 | 0 | 97.50% | 0 | 1 | 1 | 97.34% | 3 | -3 | -1 | 94.18% |
| -2 | -1 | 1 | 98.02% | 0 | 2 | -1 | 97.54% | 3 | -3 | 0 | 96.10% |
| -2 | 0 | -1 | 97.22% | 0 | 2 | 0 | 97.44% | 3 | -3 | 1 | 96.70% |
| -2 | 0 | 0 | 97.44% | 0 | 2 | 1 | 97.44% | 3 | -2 | -1 | 77.64% |
| -2 | 0 | 1 | 97.58% | 0 | 3 | -1 | 97.42% | 3 | -2 | 0 | 97.64% |
| -2 | 1 | -1 | 97.64% | 0 | 3 | 0 | 97.34% | 3 | -2 | 1 | 97.66% |
| -2 | 1 | 0 | 97.48% | 0 | 3 | 1 | 97.54% | 3 | -1 | -1 | 95.20% |
| -2 | 1 | 1 | 97.70% | 1 | -3 | -1 | 96.80% | 3 | -1 | 0 | 97.62% |
| -2 | 2 | -1 | 97.40% | 1 | -3 | 0 | 58.78% | 3 | -1 | 1 | 97.92% |
| -2 | 2 | 0 | 97.40% | 1 | -3 | 1 | 97% | 3 | 0 | -1 | 97.42% |
| -2 | 2 | 1 | 97.40% | 1 | -2 | -1 | 73.98% | 3 | 0 | 0 | 97.46% |
| -2 | 3 | -1 | 97.38% | 1 | -2 | 0 | 97.58% | 3 | 0 | 1 | 97.54% |
| -2 | 3 | 0 | 97.46% | 1 | -2 | 1 | 97.76% | 3 | 1 | -1 | 97.58% |
| -2 | 3 | 1 | 97.64% | 1 | -1 | -1 | 94.96% | 3 | 1 | 0 | 97.78% |
| -1 | -3 | -1 | 92.86% | 1 | -1 | 0 | 97.26% | 3 | 1 | 1 | 97.44% |
| -1 | -3 | 0 | 28.68% | 1 | -1 | 1 | 97.64% | 3 | 2 | -1 | 97.38% |
| -1 | -3 | 1 | 93.22% | 1 | 0 | -1 | 97.22% | 3 | 2 | 0 | 97.54% |
| -1 | -2 | -1 | 83.36% | 1 | 0 | 0 | 97.56% | 3 | 2 | 1 | 97.48% |
| -1 | -2 | 0 | 89.22% | 1 | 0 | 1 | 97.56% | 3 | 3 | -1 | 97.36% |
| -1 | -2 | 1 | 97.20% | 1 | 1 | -1 | 97.44% | 3 | 3 | 0 | 97.54% |
| -1 | -1 | -1 | 95.30% | 1 | 1 | 0 | 97.54% | 3 | 3 | 1 | 97.50% |

I define a **RBFKernelGridSearch** function to search the best combination of **(C, 𝛾) and accuracy**.

The cross validation = 3 to get the average accuracy.

The 、 = -3~ 3, and get the best set (C, 𝛾)=(,) ,accuracy=98.22%.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | Accuracy |  |  | Accuracy |
| -3 | -3 | 80.82% | 1 | -3 | 97.16% |
| -3 | -2 | 89.88% | **1** | **-2** | **98.22%** |
| -3 | -1 | 49.26% | 1 | -1 | 91.46% |
| -3 | 0 | 20.60% | 1 | 0 | 31.50% |
| -3 | 1 | 78.84% | 1 | 1 | 27% |
| -3 | 2 | 35.70% | 1 | 2 | 36.08% |
| -3 | 3 | 20% | 1 | 3 | 20% |
| -2 | -3 | 80.80% | 2 | -3 | 96.94% |
| -2 | -2 | 91.76% | 2 | -2 | 98.04% |
| -2 | -1 | 49.12% | 2 | -1 | 91.56% |
| -2 | 0 | 20.60% | 2 | 0 | 32.08% |
| -2 | 1 | 78.84% | 2 | 1 | 20.64% |
| -2 | 2 | 35.82% | 2 | 2 | 36.06% |
| -2 | 3 | 20% | 2 | 3 | 20% |
| -1 | -3 | 91.86% | 3 | -3 | 96.94% |
| -1 | -2 | 96.20% | 3 | -2 | 98.12% |
| -1 | -1 | 54.62% | 3 | -1 | 91.88% |
| -1 | 0 | 20.64% | 3 | 0 | 31.44% |
| -1 | 1 | 79.08% | 3 | 1 | 20.54% |
| -1 | 2 | 35.86% | 3 | 2 | 35.96% |
| -1 | 3 | 20% | 3 | 3 | 20% |
| 0 | -3 | 96.04% |
| 0 | -2 | 97.62% |
| 0 | -1 | 90.98% |
| 0 | 0 | 30.24% |
| 0 | 1 | 33.36% |
| 0 | 2 | 36.02% |
| 0 | 3 | 20% |

**Part3:** Use linear kernel + RBF kernel together (therefore a new kernel function) and compare its performance with respect to others. You would need to find out how to use a user-defined kernel in libsvm.

I define a userDefined\_kernel. Use svm\_problem to precomputed kernels. The parameter ”isKernel” means use precomputed kernel. The result linear kernel + RBF kernel accuracy=97.24%



**Observation 1**

The larger the C, the greater the penalty, the fewer the support vectors, and the easier it is to overfitting.  
The gamma is large, it is easy to outline the hyperplane that fits the near point, and it is easy to cause overfitting.

**Observation 2**

Try linear kernel + **polynomial kernel** + RBF kernel together

polynomial kernel degree=5

The accuracy:97.24% is the same.

