

Machine Learning Hw 5 Report

tags: Machine Learning SVM Gaussian Process

0856109 資科工碩一 方偉綸

Gaussian Process

Problem 1

First, we define the kernel function, the rational quadratic kernel is shown below.

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

```
def kernel(X_m, X_n, sigma = 1, alpha = 1, l = 1):  
    d = (X_m**2) - 2*np.dot(X_m, X_n.T) + (X_n**2).flatten()  
    return (sigma**2)*(1 + d**2/(2*alpha*l))**(-alpha)
```

Next, we compute components of the covariance matrix of the joint distribution, and derive the covariance matrix.

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

Use these components to predict the result.

$$C = k(x, x) + \beta^{-1}I$$
$$\mu = k(x, x^*)^T C^{-1}y$$
$$\sigma^2 = (k(x^*, x^*) + \beta^{-1}I) - k(x, x^*)^T C^{-1}k(x, x^*)$$

```
def prediction(X_train, Y_train, X_test, noise = .2, sigma = 1, alpha = 1, l = 1):
    n_train = X_train.shape[0]
    n_test = X_test.shape[0]

    C = np.matrix(kernel(X_train, X_train, sigma, alpha, l).reshape(n_train,
n_train)) + (noise**2)*np.eye(n_train)
    k_s = np.matrix(kernel(X_train, X_test, sigma, alpha, l).reshape(n_train,
n_test))
    k_ss = np.matrix(kernel(X_test, X_test, sigma, alpha, l).reshape(n_test,
n_test))
    mean = k_s.T*np.linalg.inv(C)*np.matrix(Y_train)
    var = k_ss + noise**2 - k_s.T*np.linalg.inv(C)*k_s

    return (mean, var)
```

Gaussian process assume each prediction point is a Gaussian distribution. The means are the most confident value of each prediction, and the 95% confidence intervals are calculated by $\mu \pm 1.96\sigma$ as shown in below.

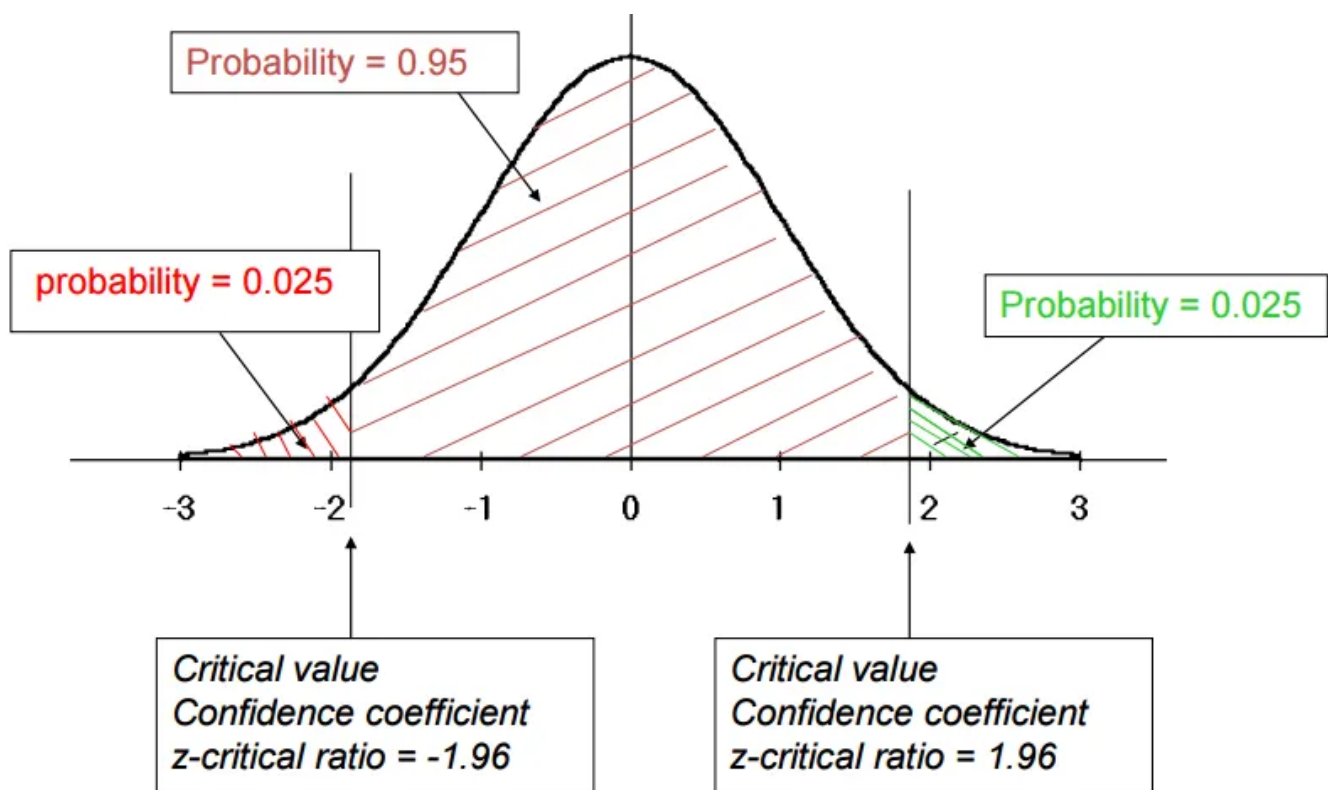


Figure 1. Confidence interval

Ref: [CONFIDENCE INTERVALS](http://makemeanalyst.com/observational-studies-and-experiments/confidence-intervals/) (<http://makemeanalyst.com/observational-studies-and-experiments/confidence-intervals/>).

Then, plot the result with `matplotlib.pyplot` .

```

X_test = np.arange(-60,60,0.5).reshape(-1,1)

m,cov = prediction(X,Y,X_test,noise,1,1,1)

plt.plot(X,Y, 'bo')
plt.plot(X_test, m, 'r')
cert = 1.96 * (np.sqrt(np.diag(cov))).reshape(-1,1)

plt.fill_between(X_test.ravel(), np.array(m + cert).ravel(), np.array(m -
cert).ravel(), alpha=0.2)
plt.show()

```

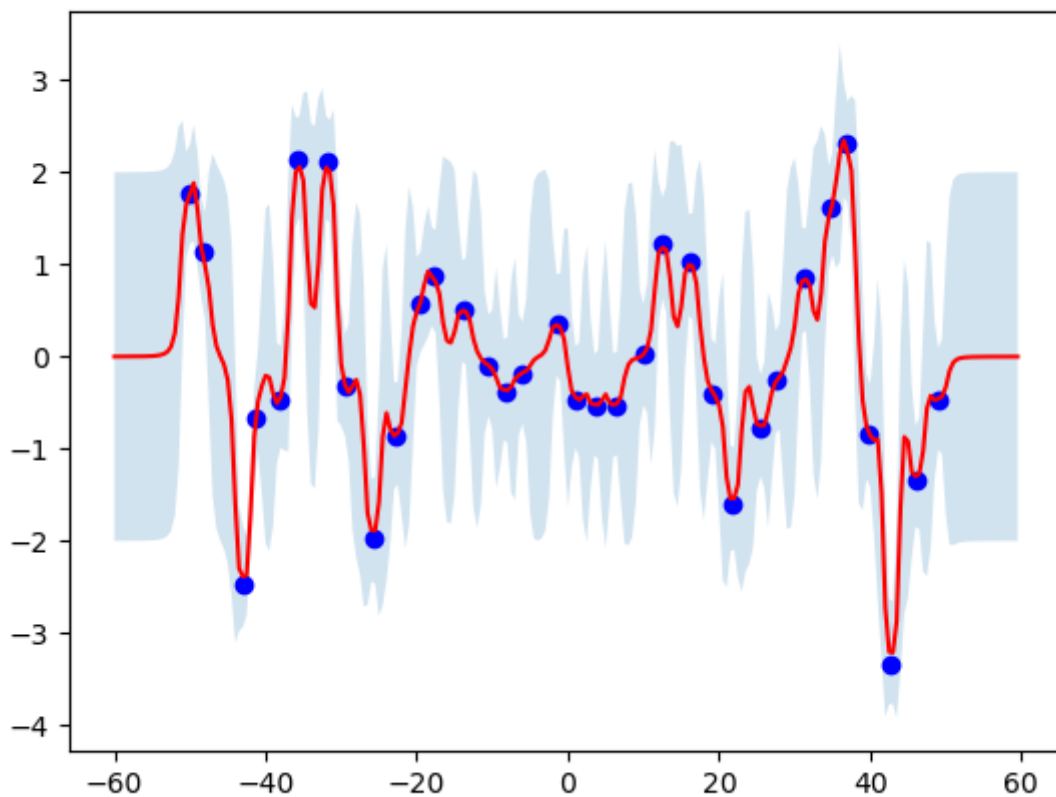


Figure 2. Result for $\sigma = 1$ and $\alpha = 1$ and $l = 1$

Problem 2

Use the `Scipy.optimize.minimize` function to optimize σ , α , and l . The log marginal likelihood function which we want to maximize (MAP) is:

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

Note that since we want to maximize the likelihood but calling minimize function, we should return a negative likelihood value to match our goal.

```
def likelihood(params):
    n = X.shape[0]
    C = np.matrix(kernel(X, X, params[0], params[1], params[2])).reshape(n, n) +
        (noise**2)*np.eye(n)

    return 1/2 * np.log(np.linalg.det(C)) + 1/2 * Y.T * np.linalg.inv(C) * Y +
        n/2 * np.log(2*np.pi)

optimal_params = minimize(likelihood, [1,1,1])
opt_sigma, opt_alpha, opt_l = optimal_params.x
opt_m, opt_cov = prediction(X,Y,X_test,noise ,opt_sigma,opt_alpha,opt_l)
```

The optimized result for each parameter and corresponding figure is shown below.

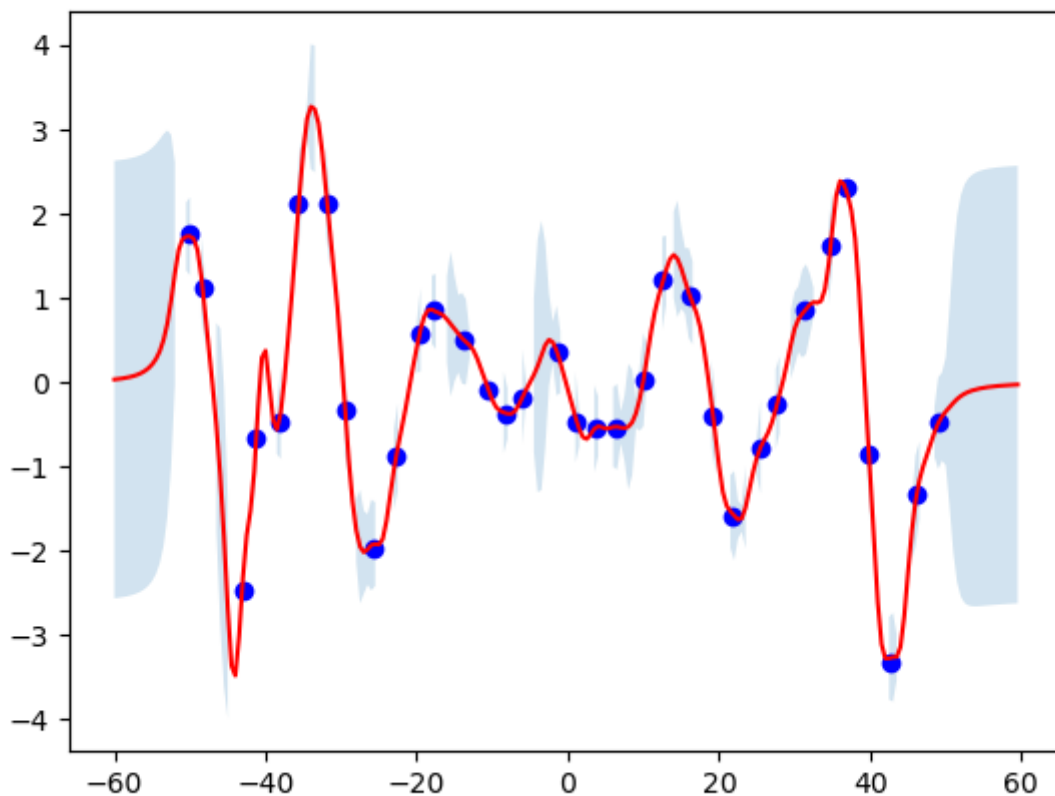


Figure 3. Result for
 $\sigma = 1.30955964$ and $\alpha = 0.56971738$ and $l = 15.48000291$

Support Vector Machine

Problem 1

In this section, we will use the `libsvm` library to classify the hand-written digits. Here are some parameters used in `libsvm`.

```
-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
-v n: n-fold cross validation mode
```

In this homework, we only use the C-SVC for SVM type. C-SVC is SVM with soft margin as show in figure 4, the original condition $t_n y(x_n) \geq 1$ is adjust to $t_n y(x_n) \geq 1 - \xi_n$ where $\xi_n = |t_n - y(x_n)|$. Thus the optimization problem now is to minimize

$$\frac{1}{2} \|w\|^2 + C \sum_{n=1}^N \xi_n$$

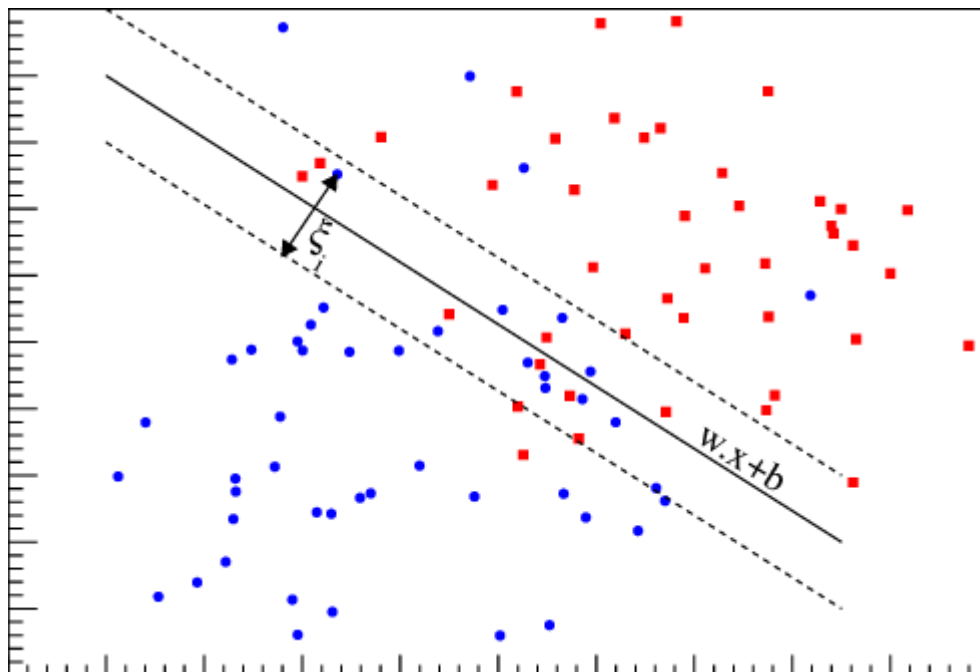


Figure 4. SVM with soft margin

Ref: [Researchgate \(https://www.researchgate.net/figure/Soft-margin-SVM-showing-the-solid-maximal-margin-hyper-plane-and-dashed-functional_fig1_313760697\)](https://www.researchgate.net/figure/Soft-margin-SVM-showing-the-solid-maximal-margin-hyper-plane-and-dashed-functional_fig1_313760697)

```
-t kernel_type : set type of kernel function (default 2)
    0 -- linear
    1 -- polynomial
    2 -- radial basis function
    3 -- sigmoid
```

And equations of each type of kernel are shown below:

linear : $K(x_i, x_j) = x_i^T x_j$
polynomial : $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$
radius basis function(RBF) : $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$
sigmoid : $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

All of above mention parameters could be defined by these option:

```
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-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)
-wi weight: set the parameter C of class i to weight*C, for C-SVC (default 1)
```

First we compare the performance of different kernel functio.

```
kernel_function_name = ['linear', 'polynomial', 'RBF']
for i in range(3):
    print("Kernel function: {}".format(kernel_function_name[i]), end="\n\t")
    t0 = time.time()
    model = svm_train(training_labels, training_data, '-q -t ' + str(i))
    svm_predict(testing_labels, testing_data, model)
    print("\tTraining time: {}s".format(time.time() - t0))
```

The result is shown below, the linear kernel has the fastest computing time while remaining high accuracy; the polynomial kernel function take the longest time but output a terrible accuracy. However, these result is based on default parameters, we will tune these parameters in the next problem.

```
Kernel function: linear
    Accuracy = 95.08% (2377/2500) (classification)
    Training time: 8.231273174285889s
Kernel function: polynomial
    Accuracy = 34.68% (867/2500) (classification)
    Training time: 82.04257845878601s
Kernel function: RBF
    Accuracy = 95.32% (2383/2500) (classification)
    Training time: 22.45405340194702s
```

Problem 2

In this problem, we try to tune the parameters of each kernel function to get the best performance. We use grid search to find the C , γ , r and d resulting in the highest accuracy. For C and γ which should greater than 0, we try 10 values, both are from 2^{-5} to 2^5 ; for r we try 0 and 1; for d we try 2,3, and 4.

```

for c in range(-5,5,1):
    params = "-q -v 5 -c {}".format(2**c)
    acc = svm_train(training_labels, training_data, params + " -t 0")
    linear_kernel_results.append((acc, 2**c))

for gamma in range(-5,5,1):
    params += " -g {}".format(2**gamma)
    acc = svm_train(training_labels, training_data, params + " -t 2")
    RBF_kernel_results.append((acc, 2**c, 2**gamma))
    for d in range(2,5,1):
        for r in range(2):
            params += " -d {} -r {}".format(d,r)
            acc = svm_train(training_labels, training_data, params + " -t 1")
            polynomial_kernel_results.append((acc, 2**c, 2**gamma, d, r))

```

We loop through every values and store the (accuracy, parameters) pair into list. After a huge time of calculating, we search for the highest accuracy pair.

```

linear_kernel_results.sort(key = lambda x: x[0], reverse = True)
polynomial_kernel_results.sort(key = lambda x: x[0], reverse = True)
RBF_kernel_results.sort(key = lambda x: x[0], reverse = True)

print(linear_kernel_results[0])
print(polynomial_kernel_results[0])
print(RBF_kernel_results[0])

```

Table 1. Optimized parameters of each kernel

kernel	C	γ	d	r	accuracy
linear	0.03125	-	-	-	96.96
polynomial	0.125	0.125	2.0	1.0	98.32
RBF	4.0	0.03125	-	-	98.60

Besides, with stored (accuracy, parameters) pair we could visualze the reationship between parameters and accuracy.

As shown in Figure 5. we found that the accuracy of linear kernel is high at about 0, and quickly decrease with C increasing.

```

plt.ylabel('accuracy')
plt.xlabel('C')
plt.title('Relationship between C and accuracy for linear kernel')
linear_kernel_results.sort(key = lambda x: x[1], reverse = True)
plt.plot([c for acc,c in linear_kernel_results], [acc for acc,c in linear_kernel_results], label= 'linear')

```

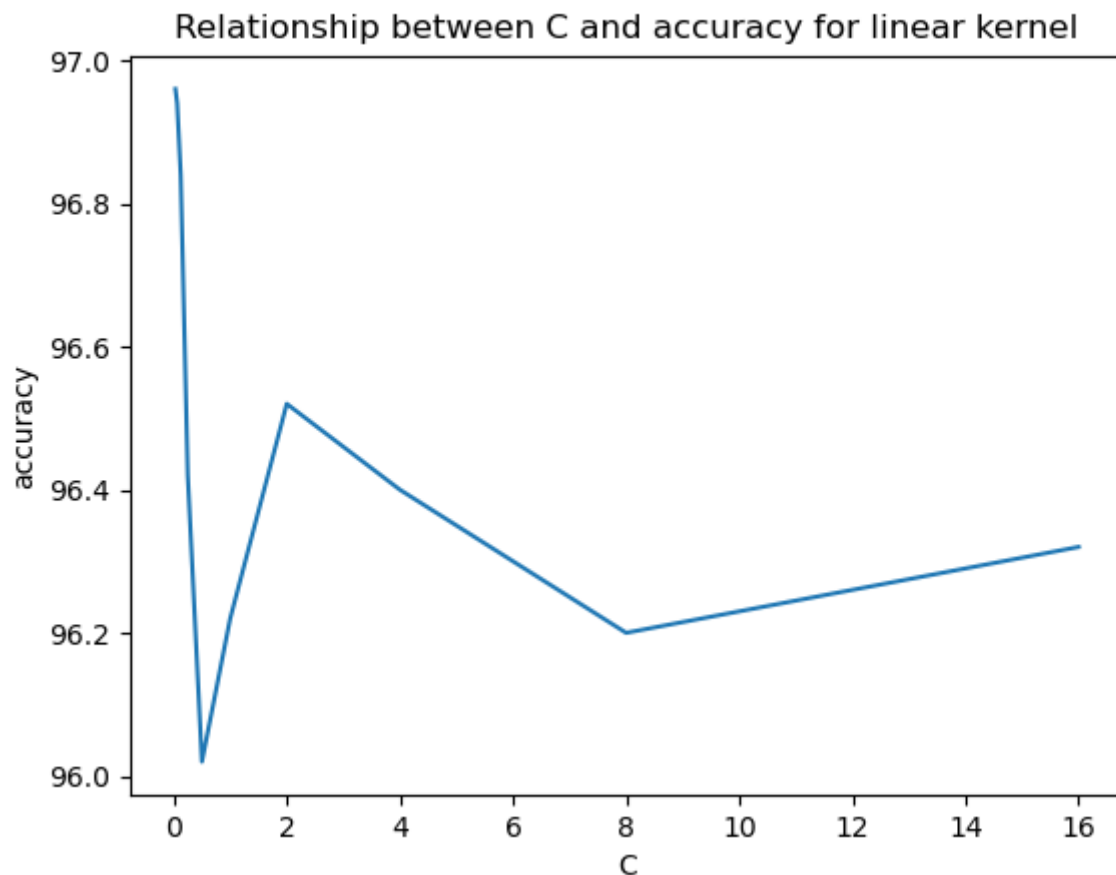


Figure 5. Relation between C and accuracy for linear kernel

For RBF kernel, if we fixed the γ , accuracy is initially low for $\gamma < 2$ and rapidly increase to steady state. as shown in Figure 6. And if we fixed the C , the accuracy will oscillate with γ increase.

```
plt.ylabel('accuracy')
plt.xlabel('C')
plt.title('Relationship between C and accuracy for RBF kernel')
for gamma in range(-5,6,1):
    tmp = [i for i in RBF_kernel_results if i[2] == 2**gamma]
    tmp.sort(key = lambda x: x[1], reverse = True)
    plt.plot([c for acc, c, g in tmp], [acc for acc, c, g in tmp], label='RFB
gamma={}'.format(2**gamma))
plt.legend()

plt.ylabel('accuracy')
plt.xlabel('Y')
plt.title('Relationship between Y and accuracy for RBF kernel')
for c in range(-5,6,1):
    tmp = [i for i in RBF_kernel_results if i[1] == 2**c]
    tmp.sort(key = lambda x: x[2], reverse = True)
    plt.plot([g for acc,c,g in tmp], [acc for acc,c,g in tmp], label='RFB c=
{}'.format(2**c))
plt.legend()
```

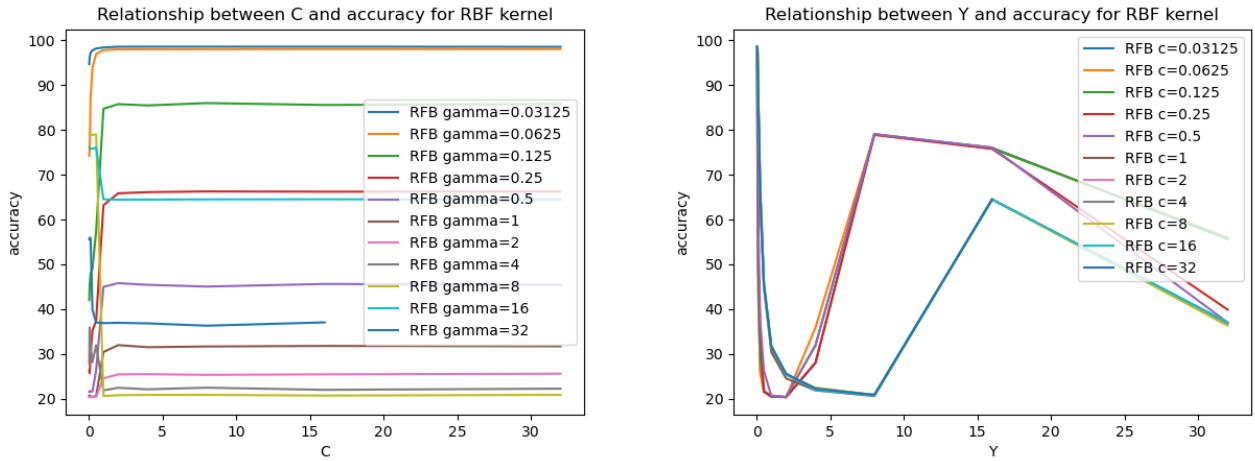



Figure 6. Relationship between C , γ and accuracy

As shown in Figure 7, polynomial has a more stable and higher accuracy compare to linear and RBF kernel. Vary in C has little influenced on accuracy except for $C \rightarrow 0$ has a relatively lower accuracy. For $d = 4, r = 1$ the variance in γ is more observable than other (d, r) .

```
plt.subplots_adjust(hspace=.3)
for gamma in range(-5,6,1):
    for d in range(2,5,1):
        for r in range(2):
            tmp = [i for i in polynomial_kernel_results if i[2] == 2**gamma and
i[3] == d and i[4] == r]
            plt.subplot(3,2,(d-2)*2+(r+1))
            plt.title("d={} r={}".format(d, r))
            plt.ylabel('accuracy')
            plt.ylim(95,99)
            plt.xlabel('C')
            tmp.sort(key = lambda x: x[1], reverse = True)
            plt.plot([c for acc, c, g, d, r in tmp], [acc for acc, c, g, d, r in
tmp], label='RFB gamma={}'.format(2**gamma))

plt.subplots_adjust(hspace=.3)
for c in range(-5,6,1):
    for d in range(2,5,1):
        for r in range(2):
            tmp = [i for i in polynomial_kernel_results if i[1] == 2**c and i[3]
== d and i[4] == r]
            plt.subplot(3,2,(d-2)*2+(r+1))
            plt.title("d={} r={}".format(d, r))
            plt.ylabel('accuracy')
            plt.ylim(95,99)
            plt.xlabel('Y')
            tmp.sort(key = lambda x: x[2], reverse = True)
            plt.plot([g for acc, c, g, d, r in tmp], [acc for acc, c, g, d, r in
tmp], label='RFB c={}'.format(2**c))
```

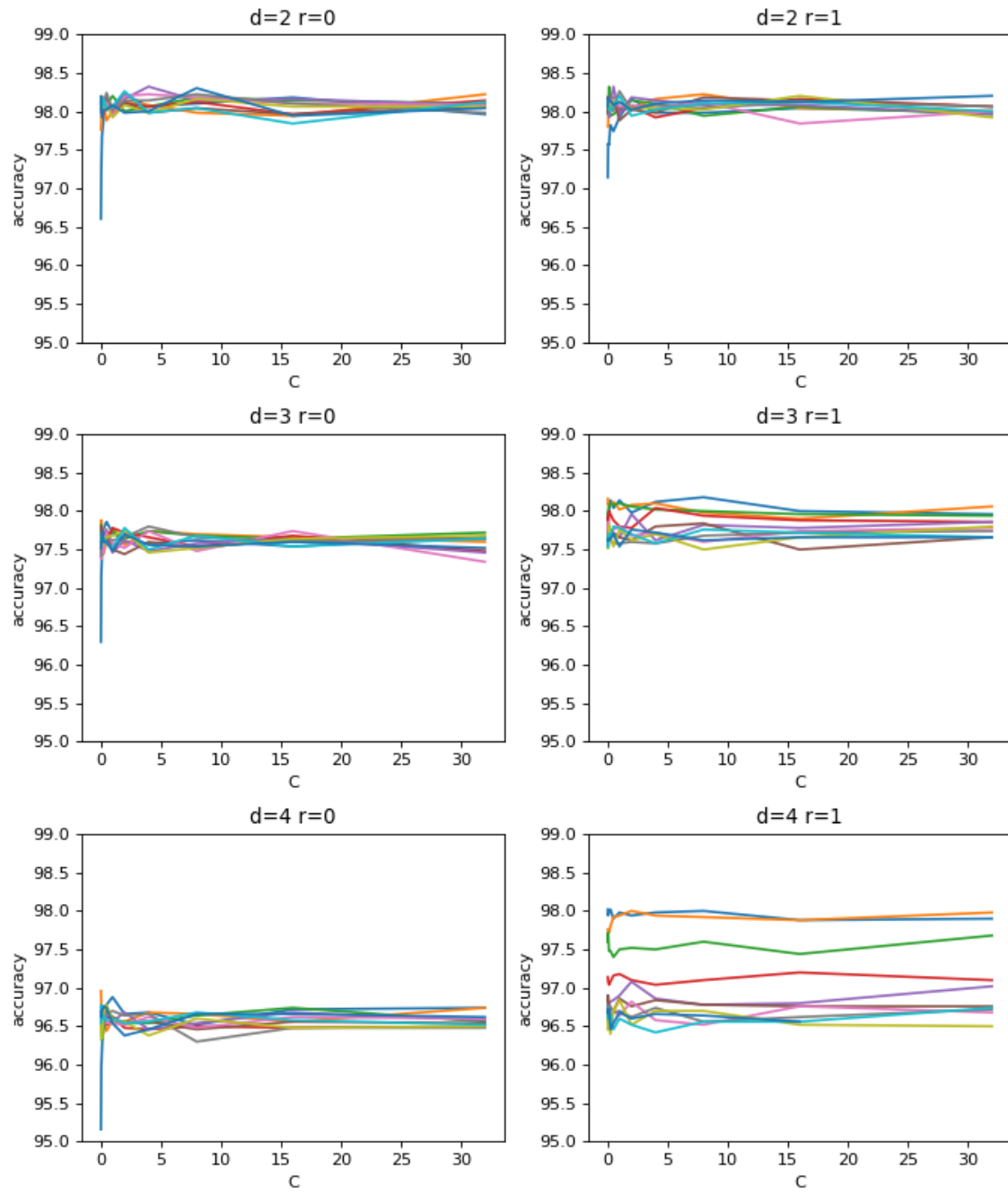


Figure 7. Relationship between C and accuracy considering different r and d

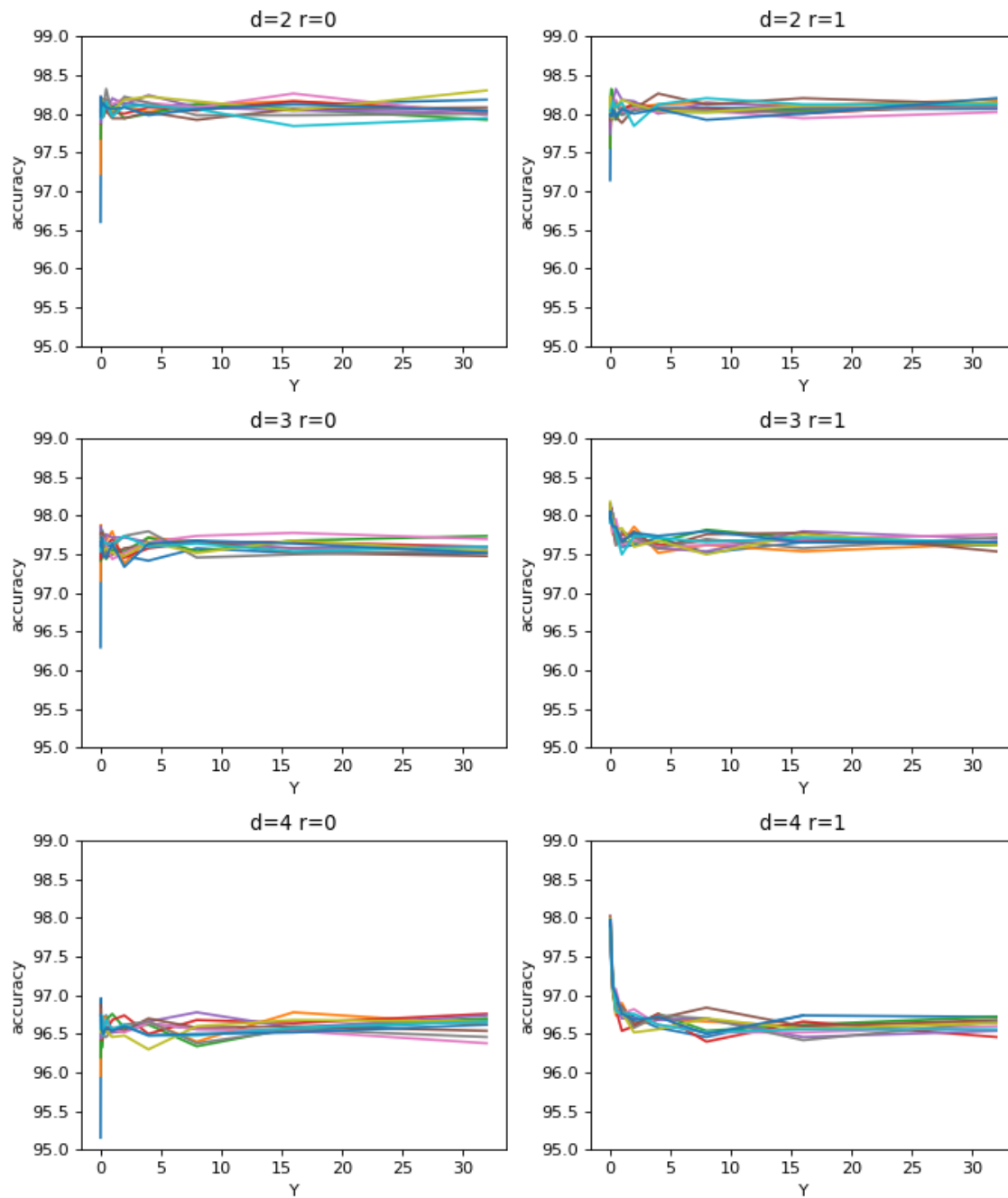


Figure 8. Relationship between γ and accuracy considering different r and d

Problem 3

To use precomputed kernel, you must include sample serial number as the first column of the training and testing data.

We first compute the linear kernel $K(x_i, x_j) = x_i^T x_j$ and the RBF kernel $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$ after adding them up, we use `numpy.hstack` to stack the custom kernel with serial number. Perform these procedure both for training data and testing data. Note that we use the optimal parameter γ from the previous problem that is 0.003125. Also, we use `scipy.spatial.pdist(X, 'sqeuclidean')` to calculate the distance $\|x_i - x_j\|^2$.

```
training_data_size = training_data.shape[0]
testing_data_size = testing_data.shape[0]
gamma = 0.03125

t0 = time.time()

linear_kernel_training = np.dot(training_data, training_data.T)
RBF_kernel_training = squareform(np.exp(- gamma * pdist(training_data,
'sqeuclidean'))))
linear_kernel_testing = np.dot(training_data, training_data.T)
RBF_kernel_testing = squareform(np.exp(- gamma * pdist(testing_data,
'sqeuclidean'))))

training_kernel = np.hstack((np.arange(1, training_data_size +
1).reshape((-1,1)), np.add(linear_kernel_training, RBF_kernel_training)))
testing_kernel = np.hstack((np.arange(1, testing_data_size + 1).reshape((-1,1)),
np.add(linear_kernel_testing, RBF_kernel_testing)))

model = svm_train(training_labels, training_kernel, '-q -t 4')
svm_predict(testing_labels, testing_kernel, model)
print("\tTraining time: {}s".format(time.time() - t0))
```

The result is much lower than original kernel.

```
Accuracy = 13.72% (343/2500) (classification)
Training time: 91.89872431755066s
```

[Original page](https://hackmd.io/@warrenpig/SkdpdvpjU) (<https://hackmd.io/@warrenpig/SkdpdvpjU>).

[GitHub Repo](https://github.com/Warrenww/Machine-Learning/tree/master/hw%205) (<https://github.com/Warrenww/Machine-Learning/tree/master/hw%205>).