

2018 ML HW5 Support Vector Machine

0756110 李東霖

This document made by HackMD, you can view here <https://hackmd.io/s/B1JFyLrRX>
(<https://hackmd.io/s/B1JFyLrRX>)

- 2018 ML HW5 Support Vector Machine
 - C-SVC (soft-margin)
 - linear 、 polynomial 、 RBF kernel
 - Prepare dataset for libsvm
 - Grid search and Cross-validation
 - User define kernel (linear + RBF)
 - Train model and Test data
 - Comparison
 - Comparison with user define kernel

C-SVC (soft-margin)

$$\min_{w,b,\xi_n} (C \sum_{n=1}^N \xi_n + 0.5 ||w||^2)$$

$$\xi_n = |t_n - y(X_n)|, t_n(w^T \phi(X_n) + b) \geq 1 - \xi_n$$

從優化目標可以看出 參數 C 的作用是決定 slack ξ_n 的多寡

- 當 C 較大，代表 slack 影響大
 - 要讓 slack 盡可能小才能靠近優化目標
 - 較少點在 margin 內
- 當 C 較小，代表 slack 影響小
 - slack 大一點也不會對優化目標有太大傷害
 - 較多點在 margin 內
- libsvm training options

-s svm_type : set type of SVM (default 0)
0 -- C-SVC

-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)

linear 、 polynomial 、 RBF kernel

- linear

$$K(u, v) = u^T v$$

- polynomial

$$K(u, v, d, \gamma, coef0) = (\gamma u^T v + coef0)^d$$

-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)

- RBF

$$K(u, v, \gamma) = \exp(-\gamma |u - v|^2)$$

-g gamma : set gamma in kernel function (default 1/num_features)

Prepare dataset for libsvm

libsvm 採用稀疏矩陣 (sparse matrix)，來儲存資料
剛好在這一次處理的資料是手寫圖片，容易有 0 (全黑) 的情況出現

- in csv

```
label index:value index:value ...  
label index:value index:value ...
```

- in python struct

```
label = [1,2]  
data = [{1:2,3:1},{3:2,10:1}]
```

- My code

```
def sparse_matrix(x, iskernel=False):
    row = x.shape[0]
    col = x.shape[1]
    idx_offset = 1
    if (iskernel):
        x = np.append(np.linspace(1,row,row), x).reshape(col+1,row).T
        idx_offset = 0
    x = [{idx+idx_offset:x[i][idx] \
        for _,idx in np.ndenumerate(np.argwhere(x[i]!=0))} \
        for i in range(x.shape[0])]
    return x

X_train = np.genfromtxt('X_train.csv', delimiter=',')
T_train = np.genfromtxt('T_train.csv', delimiter=',')
x_train = sparse_matrix(X_train)
y_train = list(T_train)

X_test = np.genfromtxt('X_test.csv', delimiter=',')
T_test = np.genfromtxt('T_test.csv', delimiter=',')
x_test = sparse_matrix(X_test)
y_test = list(T_test)
```

如果該資料是 precomputed kernel data (iskernel==True)，就要加入該筆資料的 index (x = np.append(np.linspace(1,row,row), x).reshape(col+1,row).T)

np.argwhere(x[i]!=0) 找出第 i 筆資料中不為 0 的 index，並利用 iterator 產生 dictionary

最後 idx_offset 去調整 dictionary 當中 index 的初始值 (如果是 precomputed kernel data 初始 index 是 0)

從 .csv 檔撈出原始資料並經過預處理成 libsvm 可接受資料格式 (sparse matrix)

Grid search and Cross-validation

grid search 給予不同參數不同數值，形成多個 grid，找出最好參數
cross-validation 在只有 training data 時，幫忙判斷參數的好壞

- My code

```
def GridSearchForSVM(kernel, parameter_matrix, problem, n_fold=10):
    opts = list(parameter_matrix.keys())
    opts_max = np.array([len(parameter_matrix[opts[i]]) \
                        for i in range(len(opts))])
    current_opt = np.array([0 for i in range(len(opts))])
    results = []

    optstr_init = '-t {:d} -v {:d} '.format(int(kernel),int(n_fold))

    overflow = False
    while(True):
        while (np.count_nonzero(current_opt >= opts_max)):
            reset_indicator = np.argwhere(current_opt >= opts_max)
            current_opt[reset_indicator[-1]] -= opts_max[reset_indicator[-1]]
            if (reset_indicator[-1]-1 < 0):
                overflow = True
                break;
            current_opt[reset_indicator[-1]-1] += 1

        if (overflow):
            break

        # gen option string
        optstr = optstr_init
        result = []
        for idx,para in enumerate(current_opt):
            optkey = opts[idx]
            optstr += '-' + str(optkey) + ' ' \
                + str(parameter_matrix[optkey][para]) + ' '
            result.append(parameter_matrix[optkey][para])

        # get cross-validation result
        result.append(optstr)
        result.append(svmutil.svm_train(problem, optstr))

        results.append(result)
        # try next options

        current_opt[-1] += 1

    opts += ['opt str', 'result']
    return results, opts
```

因為要找參數總共有 $k == \text{len}(\text{opts})$ 個，要設定的數值則是在 `parameter_matrix[opts[k]]`
並找到每個參數的數值個數上限 `opts_max`

使用排列組合，從 $[0, 0, \dots, 0]$ (`current_opt`) 到 `opts_max` 當作迴圈中止條件
每做完一次的 cross-validation，就準備做下一個 grid `current_opt[-1] += 1`，並且檢查是否要超過 `opts_max`，有就要進行進位

`optstr_init = '-t {:d} -v {:d} '.format(int(kernel), int(n_fold))` 設定要執行 kernel type 與要執行多少 fold 的 cross-validation
每一次的 cross-validation 都會利用 `current_opt` 與 `parameter_matrix` 去製作要給予 `svmutil.svm_train` 的參數設定字串，並放入 `result` 供之後使用
最後算出來的平均準確率 (Accuracy) 也放入 `result` 並設定好每個 `result` 每個欄位的標籤回傳回去

```
prob = svmutil.svm_problem(y_train, x_train)

linear_results, linear_options = GridSearchForSVM(0, \
    {
        'c' : [10**-5, 10**-2, 1, 10**2, 10**5]
    }, prob)
linear_table = pd.DataFrame(linear_results, columns=linear_options)
linear_table.to_csv('linear_results.csv')

poly_results, poly_options = GridSearchForSVM(1, \
    {
        'c' : [10**-2, 1, 10**2],
        'g' : [1/100, 1/300, 1/784],
        'r' : [0, 1],
        'd' : [2, 3, 4, 10]
    }, prob)
poly_table = pd.DataFrame(poly_results, columns=poly_options)
poly_table.to_csv('poly_results.csv')


rbf_results, rbf_options = GridSearchForSVM(2, \
    {
        'c' : [10**-5, 10**-2, 1, 10**2, 10**5],
        'g' : [1, 1/50, 1/100, 1/300, 1/784]
    }, prob)
rbf_table = pd.DataFrame(rbf_results, columns=rbf_options)
rbf_table.to_csv('rbf_results.csv')
```

實際去進行 grid search 與 cross validation
並把執行結果保存起來供報告使用

利用 panda DataFrame 顯示結果

```
display(linear_table.sort_values(by=['result'], ascending=False))
```

	c	opt str	result
1	0.01000	-t 0 -v 10 -c 0.01	97.06
3	100.00000	-t 0 -v 10 -c 100	96.32
4	100000.00000	-t 0 -v 10 -c 100000	96.28
2	1.00000	-t 0 -v 10 -c 1	96.24
0	0.00001	-t 0 -v 10 -c 1e-05	79.32

可以看到當 C 很小的時候，呈現 underfitting
是因為容忍 slack 容忍的太多導致分出來的線無法很好進行分類
—  李東霖

```
display(poly_table.sort_values(by=['result'], ascending=False))
```

	c	g	r	d	opt str	result
56	100.0	0.003333	0	2	-t 1 -v 10 -c 100 -g 0.0033333333333333335 -r ...	98.20
39	1.0	0.003333	1	10	-t 1 -v 10 -c 1 -g 0.0033333333333333335 -r 1 ...	98.14
48	100.0	0.010000	0	2	-t 1 -v 10 -c 100 -g 0.01 -r 0 -d 2	98.14
53	100.0	0.010000	1	3	-t 1 -v 10 -c 100 -g 0.01 -r 1 -d 3	98.14
29	1.0	0.010000	1	3	-t 1 -v 10 -c 1 -g 0.01 -r 1 -d 3	98.08
...
67	100.00	0.001276	0	10	-t 1 -v 10 -c 100 -g 0.0012755102040816326 -r ...	20.54
43	1.00	0.001276	0	10	-t 1 -v 10 -c 1 -g 0.0012755102040816326 -r 0 ...	20.52
11	0.01	0.003333	0	10	-t 1 -v 10 -c 0.01 -g 0.0033333333333333335 -r...	20.50
19	0.01	0.001276	0	10	-t 1 -v 10 -c 0.01 -g 0.0012755102040816326 -r...	20.50
35	1.00	0.003333	0	10	-t 1 -v 10 -c 1 -g 0.0033333333333333335 -r 0 ...	20.48

```
display(rbf_table.sort_values(by=['result'], ascending=False))
```

	c	g	opt str	result
21	100000.0	0.02	-t 2 -v 10 -c 100000 -g 0.02	98.64
16	100.0	0.02	-t 2 -v 10 -c 100 -g 0.02	98.58
22	100000.0	0.01	-t 2 -v 10 -c 100000 -g 0.01	98.46
11	1.0	0.02	-t 2 -v 10 -c 1 -g 0.02	98.40
17	100.0	0.01	-t 2 -v 10 -c 100 -g 0.01	98.40
12	1.0	0.01	-t 2 -v 10 -c 1 -g 0.01	98.00
...
15	100.00000	1.0	-t 2 -v 10 -c 100 -g 1	31.16
10	1.00000	1.0	-t 2 -v 10 -c 1 -g 1	30.22
0	0.00001	1.0	-t 2 -v 10 -c 1e-05 -g 1	20.56
5	0.01000	1.0	-t 2 -v 10 -c 0.01 -g 1	20.50

可以看到當 γ 參數 g 越大的時候 overfitting 的現象越明顯，導致準確率異常的低
這是因為 RBF 本身是一個高斯函數，當 g 大的時候代表 covariance 越小
使得高斯分佈非常的尖，因此容易過度擬合到訓練資料

— 🧑 李東霖

User define kernel (linear + RBF)

使用 libsvm 提供的 precomputed kernel 功能
將資料準備成如下的格式

```
New training instance for xi:

<label> 0:i 1:K(xi,x1) ... L:K(xi,xL)

New test instance for xj:

<label> 0:? 1:K(xj,x1) ... L:K(xj,xL)

# problem option
  isKernel = True

# train option
  -t 4 precomputed kernel
```

```

def linear_RBF_kernel_2(u, v, gamma=0.01):
    design_x = np.matmul(u, v.T)
    rbf_design_x = np.sum(u**2, axis=1)[:,None] \
        + np.sum(v**2, axis=1)[None,:] \
        - 2*design_x
    rbf_design_x = np.abs(rbf_design_x) * -gamma
    rbf_design_x = np.exp(rbf_design_x)
    design_x = design_x + rbf_design_x
    return design_x

x_train_precomputed = linear_RBF_kernel_2(X_train, X_train)
x_train_precomputed = sparse_matrix(x_train_precomputed, iskernl=True)
x_test_precomputed = linear_RBF_kernel_2(X_test, X_train)
x_test_precomputed = sparse_matrix(x_test_precomputed, iskernl=True)

```

linear 計算較為簡單，直接使用矩陣相乘就可以得到

RBF 的計算要使用一些小技巧 $|u - v|^2 = u^2 + v^2 - 2 * uv^T$ 去避免運算速度較慢的 for 迴圈
最終將 linear 與 RBF 結果相加成為 precomputed data，之後做成 sparse matrix

一樣也進行 grid search 與 cross-validation

```

prob_precomputed = svmutil.svm_problem(
    y_train,
    x_train_precomputed,
    isKernel=True)

linear_rbf_results, linear_rbf_options = GridSearchForSVM(4, \
    {
        'c' : np.logspace(-6,6,10)
    }, prob_precomputed)
linear_rbf_table = pd.DataFrame(
    linear_rbf_results,
    columns=linear_rbf_options)
display(linear_rbf_table.sort_values(by=['result'], ascending=False))

```

	c	opt str	result	
3	0.010000	-t 4 -v 10 -c 0.01	97.06	
4	0.215443	-t 4 -v 10 -c 0.21544346900318823	96.56	
8	46415.888336	-t 4 -v 10 -c 46415.888336127726	96.46	

Train model and Test data

把前面取得的最好參數進行模型的訓練，並拿測試資料進行測試

```
m = {}
p_label = {}
p_acc = {}
p_val = {}
train_time = {}
test_time = {}

# best parameters from grid+cross
parastrs = {
    'linear' : '-t 0 -c 0.01',
    'polynomial' : '-t 1 -c 100 -g 0.0033333333333333335 -r 0 -d 2',
    'RBF' : '-t 2 -c 100000 -g 0.02',
}
parastrs_precomputed = {
    'linear+RBF' : '-t 4 -c 0.01',
}

# start train and testing
for kernel_type, opts in parastrs.items():
    tic()
    m[kernel_type] = svmutil.svm_train(y_train, x_train, opts)
    train_time[kernel_type] = toc()
    tic()
    p_label[kernel_type], p_acc[kernel_type], p_val[kernel_type] = \
    svmutil.svm_predict(y_test, x_test, m[kernel_type])
    test_time[kernel_type] = toc()
    print('kernel type : {} , acc : {} '.format(
        kernel_type, p_acc[kernel_type]))

for kernel_type, opts in parastrs_precomputed.items():
    tic()
    m[kernel_type] = svmutil.svm_train(y_train, x_train_precomputed, opts)
    train_time[kernel_type] = toc()
    tic()
    p_label[kernel_type], p_acc[kernel_type], p_val[kernel_type] = \
    svmutil.svm_predict(y_test, x_test_precomputed, m[kernel_type])
    test_time[kernel_type] = toc()
    print('kernel type : {} , acc : {} '.format(
        kernel_type, p_acc[kernel_type]))
```

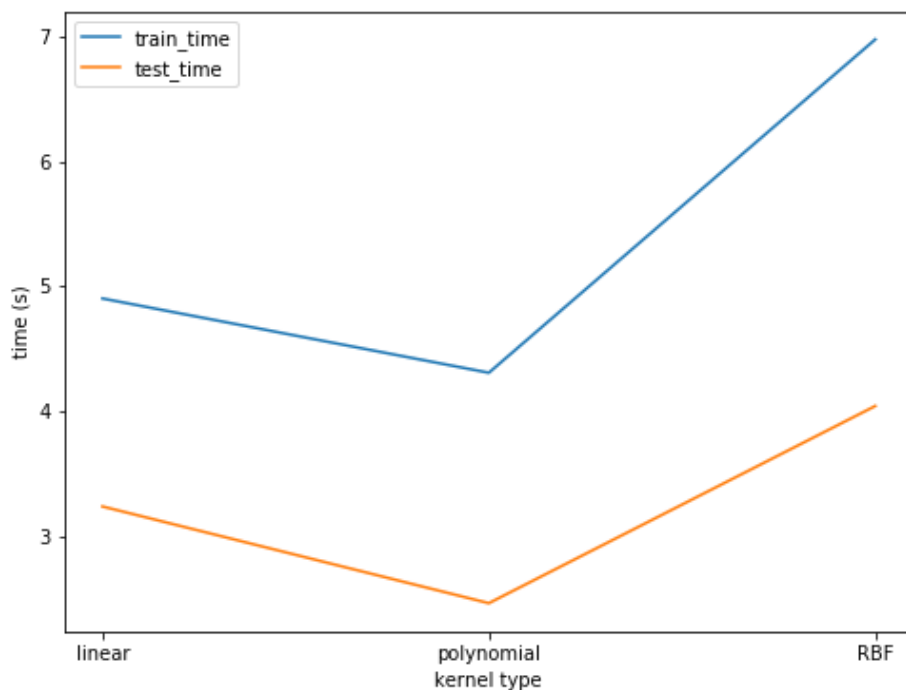
這邊因為 precomputed kernel data 用的資料與其他 kernel 不一樣，因此分開來處理
中間的 tic() toc() 是自己寫來計算時間用的，以供後面的比較

這邊直接利用 svmutil 當中的 svm_train 、 svm_predict <，來取得訓練出來的模型與測試出的準確率與各項參數。

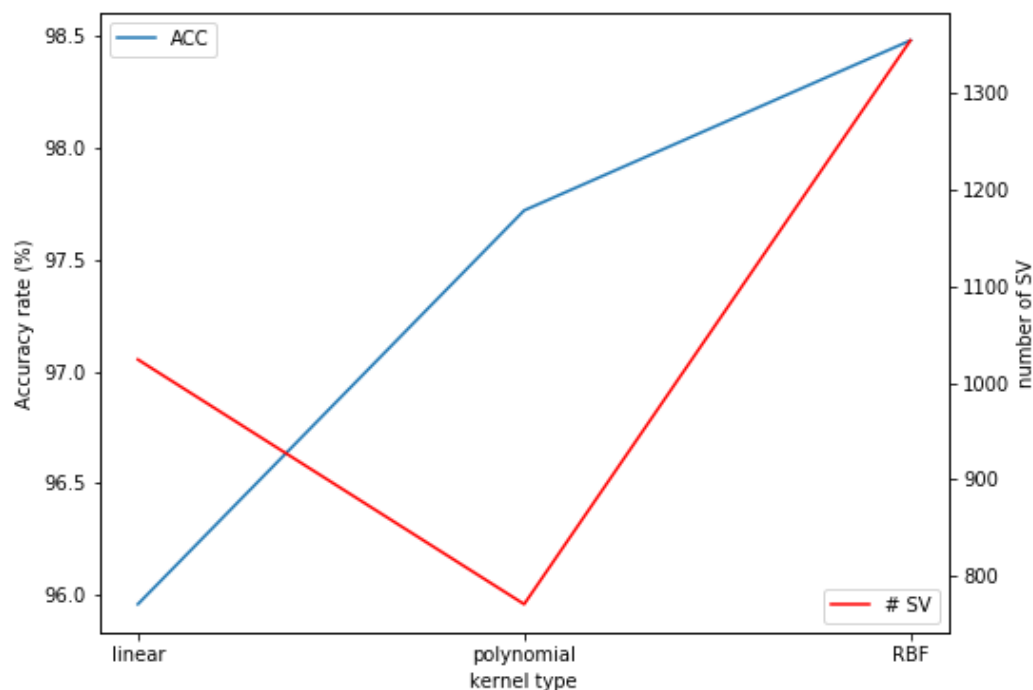
Accuracy = 95.96% (2399/2500) (classification)
kernel type : linear , acc : (95.96000000000001, 0.1256, 0.938364320909239)
Accuracy = 97.72% (2443/2500) (classification)
kernel type : polynomial , acc : (97.72, 0.0644, 0.9679859672752537)
Accuracy = 98.48% (2462/2500) (classification)
kernel type : RBF , acc : (98.48, 0.0456, 0.9772791022489454)
Accuracy = 24.8% (620/2500) (classification)
kernel type : linear+RBF , acc : (24.8, 2.406, 0.26143755235251587)

Comparison

利用 matplotlib 畫出圖表，進行比較
先比較 linear 、 polynomial 、 RBF



這邊可以看到 RBF 使用較多的訓練與測試時間，我認為是計算 kernel 較花時間
同時令人意外是 polynomial 卻使用較少的時間，理想上我認為 linear 會是最快的
— 李東霖



從圖表中可以看到 RBF 準確率最好，polynomial 次之，最差為 linear

這邊我想可以從能夠映射到的特徵空間維度可以解釋

因為 RBF 是無限的維度，所以能夠找到最好的維度去映射資料

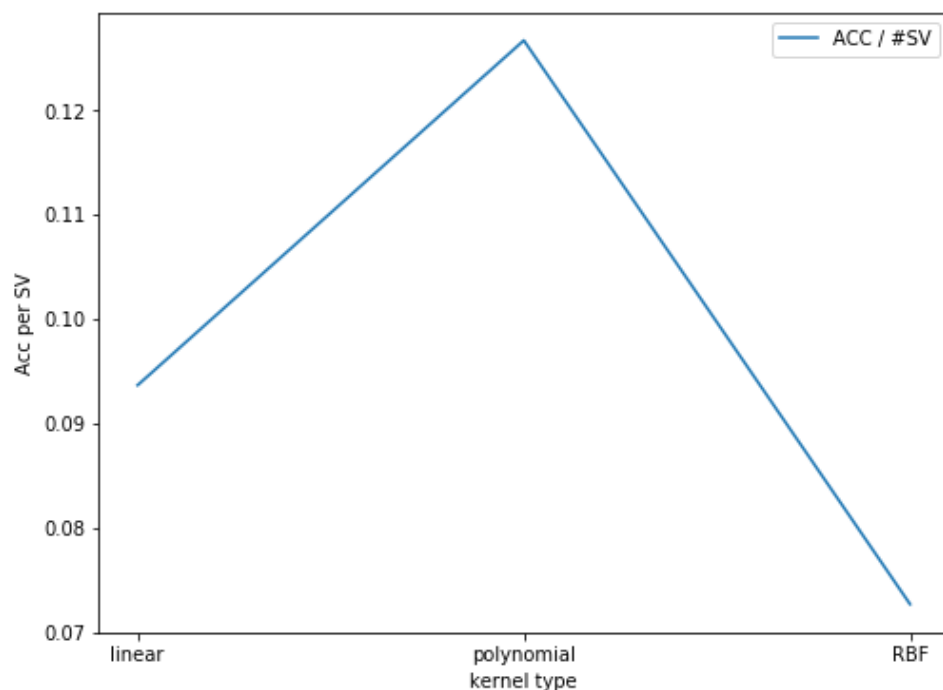
而 polynomial 的 degree 為 2，是有限的維度因此輸給 RBF

最後的 linear 就是最單純的，但可以看到準確率也有在 90% 以上

這邊呈現出另一個維度來比較三者，那就是每個訓練完的 model 需要保存多少 support vector 使用在 testing

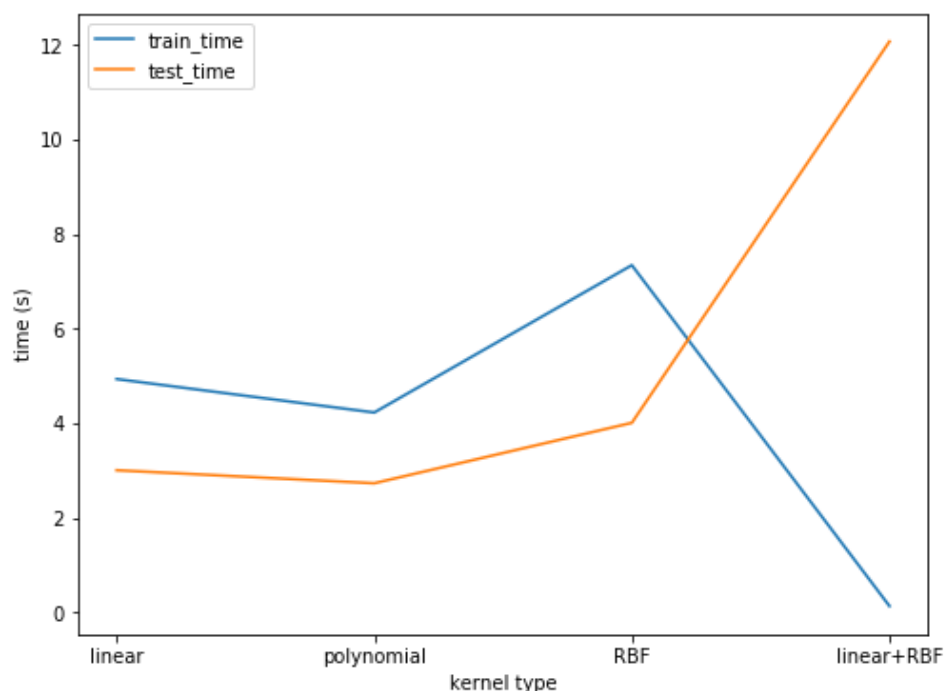
只需要 train data 的 subset 就可以進行測試是 SVM 的強項，因此這也是納入考量誰好誰壞的指標當中，因此下張圖將兩者同時進行比較，找出單位 support vector 能提供多少準確率

— 李東霖

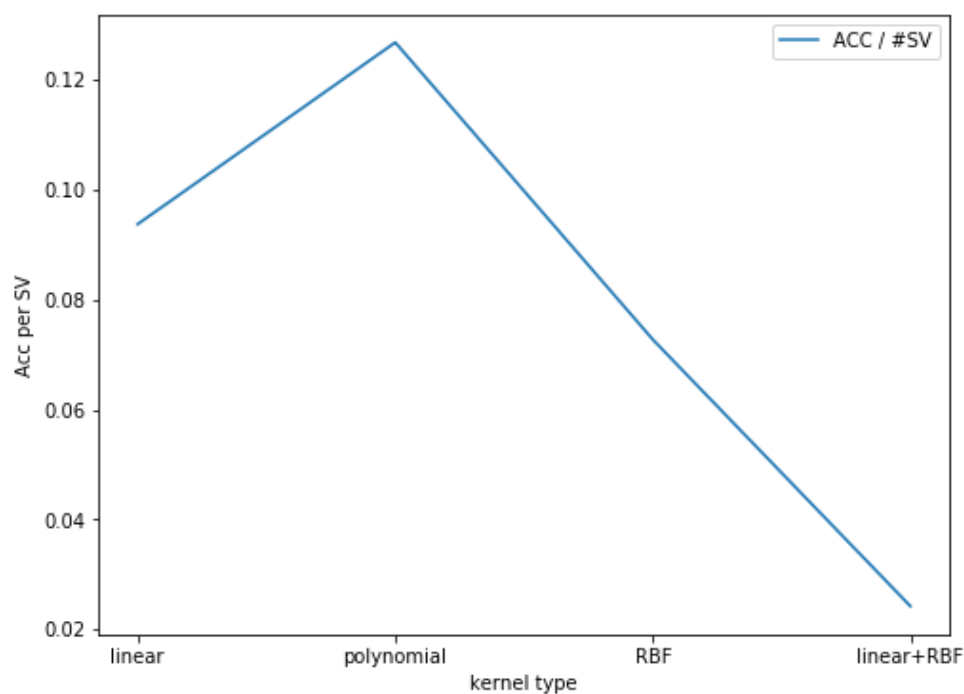
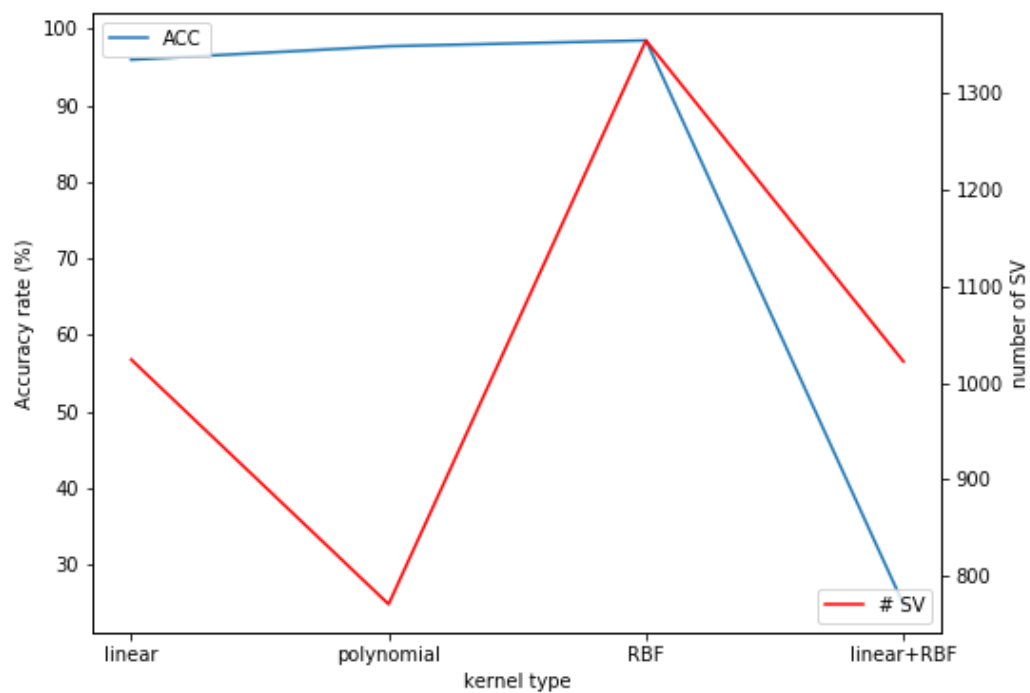


使用 Accuracy / number of SV ，試圖找出誰是最好的 model
 這邊可以看到以 polynomial kernel 的 model 是最好的
 support vector 的數量最少，都是也能提供不錯的準確率
 顯示出映射到越高維度的特徵空間，就可以需要更多的 SV 去維護
 — 李東霖

Comparison with user define kernel



可以看到 train 時間較少是因為提前計算了 kernel
 但不知如何解釋 test 時間較長
 — 李東霖



可以看到出來的結果非常不好，直到寫這報告為止，還是沒有找出原因為何在 cross-validation 時也有 90% 左右的準確率，代表資料格式有給對但不知為何到了測試階段一切都變了樣，直接成為最差的 model

— 李東霖