Support Vector Machines

- Request:
    1. Use different kernel and compare them
    2. Use C-SVM and grid search for the best parameters
    3. Use precomputed kernel
        - linear + RBF
- `libsvm` is available

Step1

Compute the user-defined kernel and convert the given training data in `libsvm` format, for example:

```
#given training data
1,2,3,4,5
2,3,4,5,6

#given training label
5
1

#libsvm-format training data
5 1:1 2:2 3:3 4:4 5:5
1 1:2 2:3 3:4 4:5 5:6

#if using precomputed kernel (i,e., request(3)),
#libsvm-format training data for index i is as below:
label_i 0:i 1:k(xi, x1) 2:k(xi, x2) 3:k(xi,x3) ...
```

For the precomputed kernel case, `isKernel` must be set to `True`.

> Referrencing the `libsvm/svm.cpp` file, and use the default `RBF` kernel in `libsvm/svm.cpp` for constructing user-defined `linear_RBF kernel` in the same way.(i.e., Transform c-style code to python-style.)
> [name=Willy Chiu]

```
def gen_libsvm_format_data(x_data, y_data, output, isKernel=False):
    x_df = pd.read_csv(x_data, header=None, dtype=str)
    y_df = pd.read_csv(y_data, header=None, dtype=str)

    for i in range(len(x_df.columns)):
        x_df[i] = str(i+1) + ':' + x_df[i].astype(str)
```

```
    if (isKernel):
        k_df = np.arange(1,len(y_df)+1)
        k_df = pd.DataFrame(k_df)
        k_df = '0:' + k_df[0].astype(str)
        x_df = pd.concat([k_df, x_df], axis = 1)

    format_df = pd.concat([y_df, x_df], axis = 1)
    format_df.to_csv(output, sep=' ', index=False, header=None)

def linear_RBF_kernel(u, v, g):
    linear_x = np.matmul(u, v.T)
    rbf_x = np.sum(u**2, axis=1)[:,None] + np.sum(v**2, axis=1)[None,:] -
2*linear_x
    rbf_x = np.abs(rbf_x) * (-g)
    rbf_x = np.exp(rbf_x)
    return linear_x + rbf_x
```

```
#preparing the precomputed kernel
print('preparing the precomputed kernel...')
x_train = np.genfromtxt('dataset/X_train.csv', delimiter=',')
x_test = np.genfromtxt('dataset/X_test.csv', delimiter=',')
x_train_precomputed = linear_RBF_kernel(x_train, x_train, 0.03125)
x_test_precomputed = linear_RBF_kernel(x_test, x_test, 0.03125)
np.savetxt('dataset/X_train_precomputed.csv', x_train_precomputed, fmt='%f',
delimiter=',')
np.savetxt('dataset/X_test_precomputed.csv', x_test_precomputed, fmt='%f',
delimiter=',')

#convert to libsvm-format
print('converting to libsvm-format...')
gen_libsvm_format_data('dataset/X_train_precomputed.csv',
'dataset/Y_train.csv', 'train_precomputed.csv', isKernel=True)
gen_libsvm_format_data('dataset/X_test_precomputed.csv', 'dataset/Y_test.csv',
'test_precomputed.csv', isKernel=True)
gen_libsvm_format_data('dataset/X_train.csv', 'dataset/Y_train.csv',
'train.csv')
gen_libsvm_format_data('dataset/X_test.csv', 'dataset/Y_test.csv', 'test.csv')

#open training data
print('opening training data...')
y_train, x_train = svmutil.svm_read_problem('train.csv')
prob = svmutil.svm_problem(y_train, x_train)
y1, x_train_precomputed = svmutil.svm_read_problem('train_precomputed.csv')
prob_precomputed = svmutil.svm_problem(y1, x_train_precomputed, isKernel=True)
```

Step2

Grid search for each kernel (include the precomputed kernel) and find the best parameters of them.

> `libsvm/tools/grid.py` provide the API to grid-search for `RBF` kernel. I tried to revise it into `./grid.py`, so that it could find the best parameters of kernels which could be taken in the `libsvm` (except sigmoid `kernel`).
>
> And the `./grid.py` take a log scale to the searching range such as `libsvm/tools/grid.py` did.
> [name=Willy Chiu]

```
# grid search for the best parameters
print('grid searching...')
grid_search = {
        'linear' : '-t 0 -log2c -5,15,2',
        'polynomial' : '-t 1 -log2c -5,15,2 -log2g -5,15,2 -log2r -3,5,2 -d
4',
        'RBF' : '-t 2 -log2c -5,15,2 -log2g -5,15,2',
        }
grid_search_precomputed = {
        'linear+RBF': '-t 4 -log2c -5,15,2'
        }
best_param = []
kernel_tab = []
kernel = []

for kernel_type, opts in grid_search.items():
    rst, tab, col = grid.find_parameters(prob, opts)
    kernel.append(kernel_type)
    kernel_tab.append(pd.DataFrame(tab, columns=col))
    best_param.append(rst)

for kernel_type, opts in grid_search_precomputed.items():
    rst, tab, col = grid.find_parameters(prob_precomputed, opts)
    kernel.append(kernel_type)
    kernel_tab.append(pd.DataFrame(tab, columns=col))
    best_param.append(rst)

for i in range(len(kernel_tab)):
    kernel_tab[i].to_csv('best param/'+kernel[i]+'.csv')
    print(kernel[i] + ':', end=' ')
    print(best_param[i])
```

- Output

```
...
[ 2821/ 2860] Cross Validation Accuracy = 96.34%
[ 2822/ 2860] Cross Validation Accuracy = 98.2%
[ 2823/ 2860] Cross Validation Accuracy = 97.7%
...
linear: [0.03125, 97.2]
polynomial: [0.125, 8192, 32, 2, 98.4]
RBF: [2048, 0.03125, 98.74]
linear+RBF: [0.03125, 97.06]
```

Step3

According the parameters form `Step2`, doing prediction.

> For saving time, I store the model after the first training of a new options and use the best parameters from grid-search result.
> [name=Willy Chiu]

```
'''
linear kernel:      Best c=0.03125, rate=97.2%
polynomial kernel:  Best c=0.125, g=8192, r=32, d=2, rate=98.4%
RBF kernel:         Best c=2048, g=0.03125, rate=98.74%
linear+RBF:         Best c=0.03125, rate=97.06
'''
# predict
print('training and predicting...')
options = {
    'linear' : '-q -t 0 -c 0.03125',
    'polynomial' : '-q -t 1 -c 0.0125 -g 8192 -r 32 -d 2',
    'RBF' : '-q -t 2 -c 2048 -g 0.03125',
}
options_precomputed = {
    'linear+RBF' : '-q -t 4 -c 0.03125',
}

m = {}
p_label = {}
p_acc = {}
p_val = {}
train_time = {}
test_time = {}
train_flag = True

y_test, x_test = svmutil.svm_read_problem('test.csv')
y1, x_test_precomputed = svmutil.svm_read_problem('test_precomputed.csv')
```

```
for kernel_type, opts in options.items():
    print('\tkernel type: {0}\n\t'.format(kernel_type), end='')
    tic()
    if (train_flag):
        m[kernel_type] = svmutil.svm_train(prob, opts)
        svmutil.svm_save_model('model/'+kernel_type+'.model', m[kernel_type])
    else:
        m[kernel_type] = svmutil.svm_load_model('model/'+kernel_type+'.model')
    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('\tresult(acc, mse, scc): {0}\n'.format(
        p_acc[kernel_type]))

for kernel_type, opts in options_precomputed.items():
    print('\tkernel type: {0}\n\t'.format(kernel_type), end='')
    tic()
    if (train_flag):
        m[kernel_type] = svmutil.svm_train(prob_precomputed, opts)
        svmutil.svm_save_model('model/'+kernel_type+'.model', m[kernel_type])
    else:
        m[kernel_type] = svmutil.svm_load_model('model/'+kernel_type+'.model')
    train_time[kernel_type] = toc()
    tic()
    p_label[kernel_type], p_acc[kernel_type], p_val[kernel_type] = \
    svmutil.svm_predict(y1, x_test_precomputed, m[kernel_type])
    test_time[kernel_type] = toc()
    print('\tresult(acc, mse, scc): {0}\n'.format(
        p_acc[kernel_type]))
```

- Output

```
predicting...
    kernel type: linear
    Accuracy = 96% (2400/2500) (classification)
    result(acc, mse, scc): (96.0, 0.1308, 0.9357489703153389)

    kernel type: polynomial
    Accuracy = 97.68% (2442/2500) (classification)
    result(acc, mse, scc): (97.68, 0.0648, 0.9677862381298715)

    kernel type: RBF
    Accuracy = 98.52% (2463/2500) (classification)
    result(acc, mse, scc): (98.52, 0.05, 0.9750879341085731)

    kernel type: linear+RBF
    Accuracy = 21.36% (534/2500) (classification)
```

```
result(acc, mse, scc): (21.36, 2.9296, 0.023906139154160982)
```

Step4

Besides the ACC rate and MSE, I also record the training time, predict time and number of support vectors for each kernel.

```
# compare result
print('visualizing...')
kernel = list(options.keys()) + list(options_precomputed.keys())
p1 = plt.bar(kernel, [test_time[i] for i in kernel], label='test_time',
alpha=0.4)
p2 = plt.bar(kernel, [train_time[i] for i in kernel], label='train_time',
alpha=0.4, bottom=[test_time[kk] for kk in kernel])
plt.ylabel('time (s)')
plt.xlabel('kernel type')
plt.legend((p1[0], p2[0]), ('test_time', 'train_time'))
plt.savefig('result/times.png')
plt.show()

fig, ytick1 = plt.subplots()
ytick2 = ytick1.twinx()

ytick2.bar(kernel,[m[i].get_nr_sv() for i in kernel], label='# SV', alpha=0.4)
ytick2.set_ylabel('number of SV')
ytick2.legend(loc='lower right')

ytick1.plot(kernel,[p_acc[i][0] for i in kernel], 'b', label='ACC')
ytick1.plot(kernel,[100*p_acc[i][1] for i in kernel], 'r', label='MSE*100')
ytick1.plot(kernel, [100 for i in kernel], 'gray', linestyle='--')
ytick1.set_ylabel('Accuracy rate (%)')
ytick1.legend(loc='upper left')
ytick1.set_xlabel('kernel type')
plt.savefig('result/results.png')
plt.show()
```

Analysis

In general, we would get higher ACC rate when the value of `c` is smaller. This reason of this circustance is we would get the smaller `slack` when setting larger value of `c`, so that overfitting would be occurred.

- Best parameter of each kernel
    - Linear

c	options	rate
0.03125	-q -t 0 -v 5 -c 0.03125	97.2
0.125	-q -t 0 -v 5 -c 0.125	96.98
0.5	-q -t 0 -v 5 -c 0.5	96.22
2.0	-q -t 0 -v 5 -c 2.0	96.28
8.0	-q -t 0 -v 5 -c 8.0	96.32
32.0	-q -t 0 -v 5 -c 32.0	96.2
128.0	-q -t 0 -v 5 -c 128.0	96.34
512.0	-q -t 0 -v 5 -c 512.0	96.34
2048.0	-q -t 0 -v 5 -c 2048.0	96.16
8192.0	-q -t 0 -v 5 -c 8192.0	96.16
32768.0	-q -t 0 -v 5 -c 32768.0	96.48

- Polynomial

c	g	r	d	options	rate
0.125	8192.0	32.0	2	-q -t 1 -v 5 -c 0.125 -g 8192.0 -r 32.0 -d 2	98.4
0.125	512.0	32.0	2	-q -t 1 -v 5 -c 0.125 -g 512.0 -r 32.0 -d 2	98.32
32768.0	2.0	2.0	2	-q -t 1 -v 5 -c 32768.0 -g 2.0 -r 2.0 -d 2	98.32
...					
0.03125	0.03125	8.0	1	-q -t 1 -v 5 -c 0.03125 -g 0.03125 -r 8.0 -d 1	95.44
0.03125	0.03125	32.0	1	-q -t 1 -v 5 -c 0.03125 -g 0.03125 -r 32.0 -d 1	95.40
0.03125	0.03125	0.5	1	-q -t 1 -v 5 -c 0.03125 -g 0.03125 -r 0.5 -d 1	95.36

- RBF

c	g	options	rate
2048.0	0.03125	-q -t 2 -v 5 -c 2048.0 -g 0.03125	98.74
2.0	0.03125	-q -t 2 -v 5 -c 2.0 -g 0.03125	98.62
8.0	0.03125	-q -t 2 -v 5 -c 8.0 -g 0.03125	98.6
...			
32768.0	2048.0	-q -t 2 -v 5 -c 32768.0 -g 2048.0	20.0
32768.0	8192.0	-q -t 2 -v 5 -c 32768.0 -g 8192.0	20.0
32768.0	32768.0	-q -t 2 -v 5 -c 32768.0 -g 32768.0	20.0

Kernel function: $e^{-\gamma||x,x_*||^2}$
According to the experiment result, when the γ becoming larger, the `ACC rate` is much better than smaller γ.

> Because the RBF kernel's formula is similar to Gaussion distribution (i.e., they are in direct proportion), and the γ is equal to $\frac{1}{2\sigma^2}$, when γ is larger hints that σ^2 is smaller. And when σ^2 is smaller hints that the PDF of Gaussion distribution is sharper, so the overfitting may occurred.
> [name=Willy Chiu]

- Linear + RBF

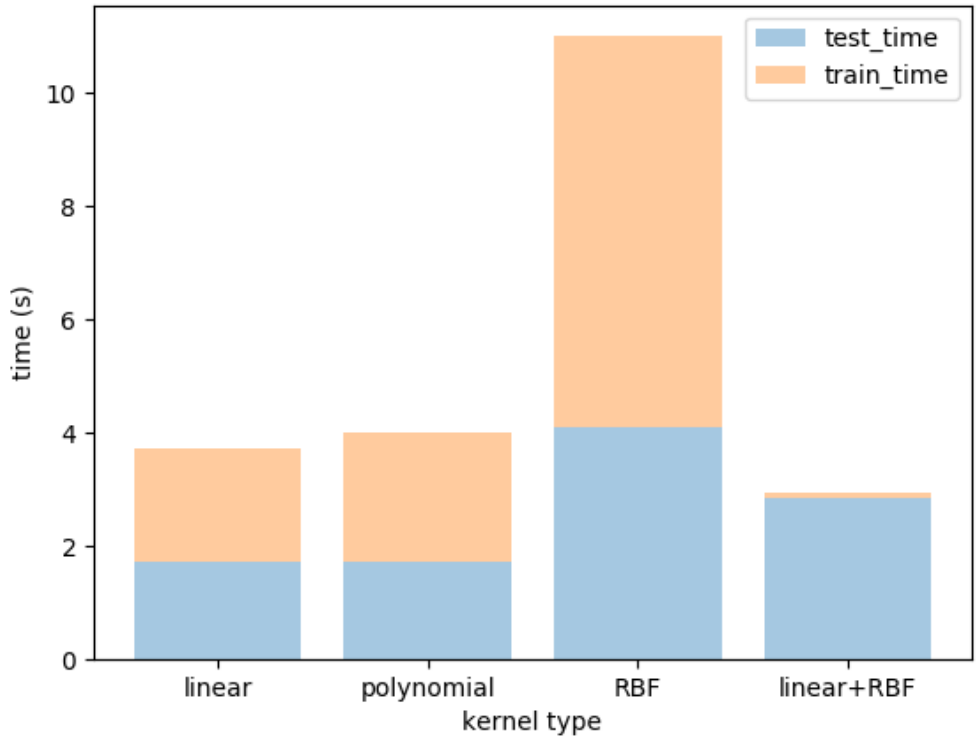
c	options	rate
0.03125	-q -t 4 -v 5 -c 0.03125	97.18
0.125	-q -t 4 -v 5 -c 0.125	97.0
32768.0	-q -t 4 -v 5 -c 32768.0	96.66
8.0	-q -t 4 -v 5 -c 8.0	96.64
128.0	-q -t 4 -v 5 -c 128.0	96.64
512.0	-q -t 4 -v 5 -c 512.0	96.62
2.0	-q -t 4 -v 5 -c 2.0	96.56
8192.0	-q -t 4 -v 5 -c 8192.0	96.5
0.5	-q -t 4 -v 5 -c 0.5	96.46
32.0	-q -t 4 -v 5 -c 32.0	96.42
2048.0	-q -t 4 -v 5 -c 2048.0	96.42

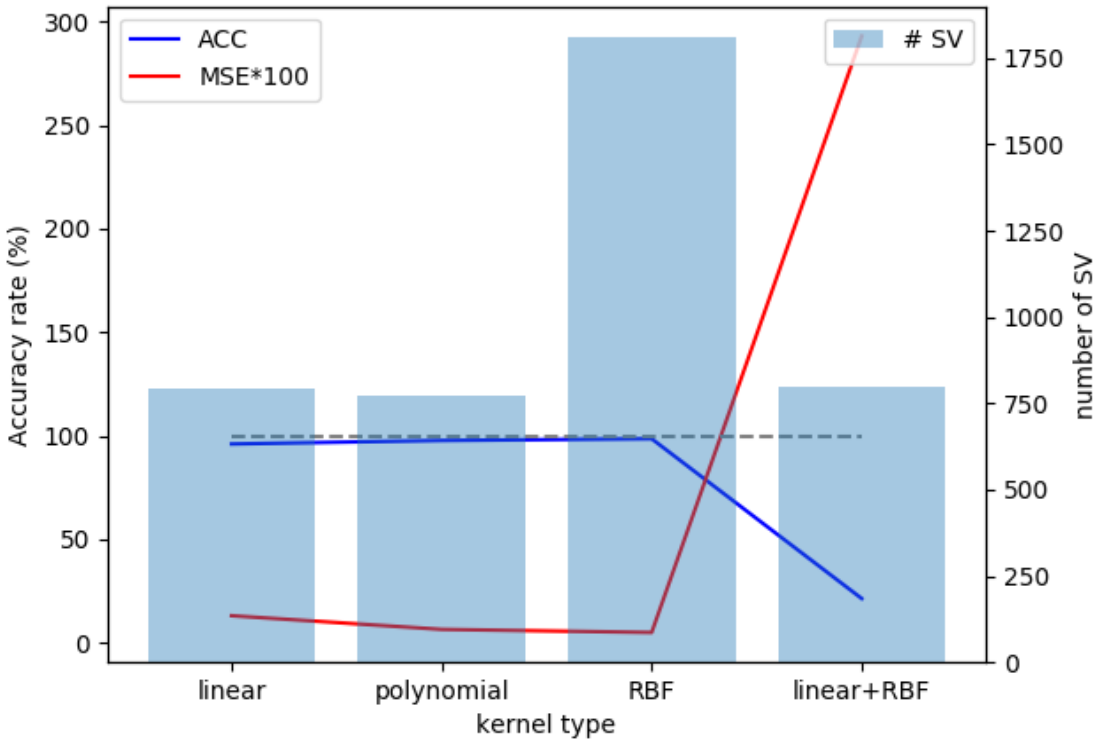
> Linear+RBF kernel seems to work well while doing cross validation, but i got such a trashlike result while predicting.

> At the first, I thought it might be wrong in the precomputed format and I tried to replace the default RBF kernel function in `libsvm` into linear+RBF function and re-run the RBF kernel (It is linear+RBF kernel now) for checking whether the predicting result is bad or not.

> But it still bad..., I don't even know why. I guess the reason is that RBF is none-linear, so when it combine to a linear kernel, the kernel funciton works badly. [name=Willy Chiu]

- Executing result of each kernel
  - RBF's trainging and predicting are the most time consuming. But it has the best ACC rate.





Referrence

- https://www.csie.ntu.edu.tw/~cjlin/libsvm/
- https://github.com/cjlin1/libsvm