ML HW5 report

資科工碩一 0756138 黃伯凱

Implement SVM by sklearn.svm.SVC :
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Kernel function for Linear, Polynomial and RBF are shown below:

Linear : $K(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}'$

Polynomial : $K(\mathbf{x}, \mathbf{x}') = (\zeta + \gamma \mathbf{x}^T \mathbf{x}')^Q$

RBF : $K(\mathbf{x}, \mathbf{x}') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2)$

Question 1:

Linear Kernel

```
tStart = time()
print("Linear Starts...")

svm = SVC(kernel='linear', gamma='auto')
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("Linear Kernel costs : %.2f sec" %(tEnd - tStart))
```

1

2

```
Linear Starts...
Precision : 95.080%
Linear Kernel costs : 4.36 sec
```

Polynomial Kernel

```
# Degree = 1
tStart = time()
print("Polynomial Starts... (Degree = 1)")

svm = SVC(kernel='poly', gamma='auto', degree=1)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("Poly Kernel costs : %.2f sec\n" %(tEnd - tStart))
```

1

2

```
# Degree = 2
tStart = time()
print("Polynomial Starts... (Degree = 2)")

svm = SVC(kernel='poly', gamma='auto', degree=2)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("Poly Kernel costs : %.2f sec\n" %(tEnd - tStart))
```

1

2

```
# Degree = 3
tStart = time()
print("Polynomial Starts... (Degree = 3)")

svm = SVC(kernel='poly', gamma='auto', degree=3)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("Poly Kernel costs : %.2f sec\n" %(tEnd - tStart))
```

1

2

```python
# Degree = 4
tStart = time()
print("Polynomial Starts... (Degree = 4)")

svm = SVC(kernel='poly', gamma='auto', degree=4)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("Poly Kernel costs : %.2f sec" %(tEnd - tStart))
```

1

2

```
Polynomial Starts...
Precision : 94.800%
Poly Kernel costs : 14.27 sec

Polynomial Starts...
Precision : 88.080%
Poly Kernel costs : 34.96 sec

Polynomial Starts...
Precision : 34.520%
Poly Kernel costs : 53.22 sec

Polynomial Starts...
Precision : 23.720%
Poly Kernel costs : 53.13 sec
```

RBF

```python
tStart = time()
print("RBF Starts...")

svm = SVC(kernel='rbf', gamma='auto')
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("RBF Kernel costs : %.2f sec" %(tEnd - tStart))
```

1

2

```
RBF Starts...
Precision : 95.320%
RBF Kernel costs : 10.74 sec
```

1. Set the model as linear, polynomial or rbf kernel, and set gamma as auto. Notice that polynomial may has different degree.
2. Calculate the accuracy by using svm.predict to predict cluster for each data.

Comparison:

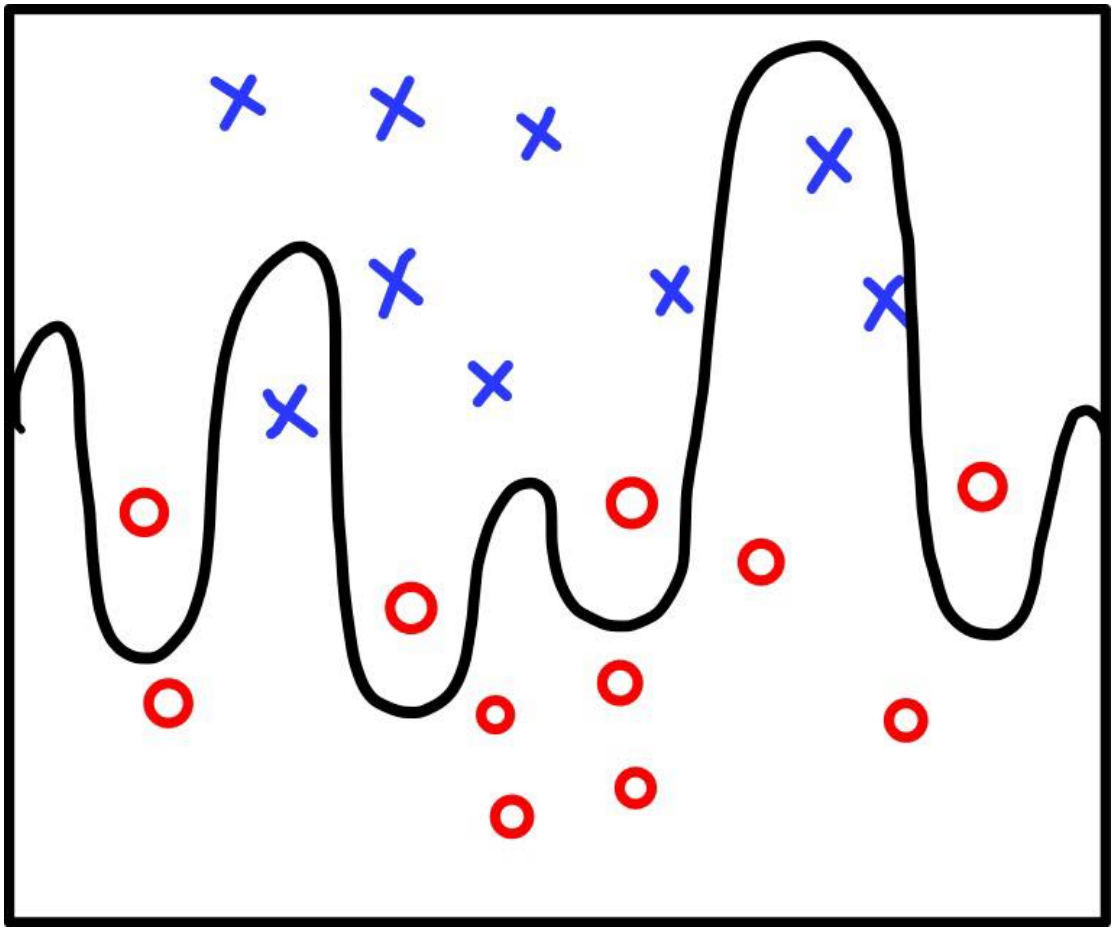
The precision order is:

rbf > linear > polynomial

The Calculation time is:

linear < rbf < polynomial

Moreover, it is obvious that the higher degree of polynomial kernel, the lower precision it has. Maybe it is because of the following situation:



Question 2:

```
# Set the parameters by cross-validation
tuned_parameters = [{'kernel': ['linear'], 'C': [8, 9, 11, 12]},
                    {'kernel': ['poly'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4],
                     'C': [8, 9, 11, 12], 'degree': [1, 2, 3]},
                    {'kernel': ['rbf'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4], 'C': [8, 9, 11, 12]}]

scores = ['precision']

tStart = time()
print("# Tuning hyper-parameters for %s" % score)
print()

clf = GridSearchCV(SVC(), tuned_parameters, cv=5,
                   scoring='%s_macro' % score)
clf.fit(X_test, T_test)

print("Best parameters set found on development set:")
print()
print(clf.best_params_)
print()
print("Grid scores on development set:")
print()
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"
          % (mean, std * 2, params))
print()

tEnd = time()
minute = (int)((tEnd-tStart)/60)
second = (int)((tEnd-tStart)%60)
print("Totally takes %d min %f sec" %(minute, second))
```

1. Set all the parameters for the SVM model
2. Grid search according to the previous parameters
3. Print the best accuracy and its parameters
4. Print all the other results

I have tried several different parameters (I will only put some of them here, otherwise it is too much):

```
tuned_parameters = [{'kernel': ['linear'], 'C': [8, 9, 11, 12]},
                    {'kernel': ['poly'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4],
                     'C': [8, 9, 11, 12], 'degree': [1, 2, 3]},
                    {'kernel': ['rbf'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4], 'C': [8, 9, 11, 12]}]
```

```
tuned_parameters = [{'kernel': ['linear'], 'C': [3, 4, 6, 7]},
                    {'kernel': ['poly'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4],
                     'C': [3, 4, 6, 7], 'degree': [1, 2, 3]},
                    {'kernel': ['rbf'], 'gamma': [1, 1e-1, 1e-2, 1e-3, 1e-4], 'C': [3, 4, 6, 7]}]
```

```
Best parameters set found on development set:

{'C': 2, 'gamma': 0.01, 'kernel': 'rbf'}

Grid scores on development set:

0.945 (+/-0.029) for {'C': 2, 'kernel': 'linear'}
0.945 (+/-0.029) for {'C': 5, 'kernel': 'linear'}
0.945 (+/-0.029) for {'C': 15, 'kernel': 'linear'}
0.945 (+/-0.029) for {'C': 20, 'kernel': 'linear'}
0.945 (+/-0.029) for {'C': 2, 'degree': 1, 'gamma': 1, 'kernel': 'poly'}
0.951 (+/-0.025) for {'C': 2, 'degree': 1, 'gamma': 0.1, 'kernel': 'poly'}
0.956 (+/-0.027) for {'C': 2, 'degree': 1, 'gamma': 0.01, 'kernel': 'poly'}
0.946 (+/-0.025) for {'C': 2, 'degree': 1, 'gamma': 0.001, 'kernel': 'poly'}
0.894 (+/-0.052) for {'C': 2, 'degree': 1, 'gamma': 0.0001, 'kernel': 'poly'}
0.974 (+/-0.019) for {'C': 2, 'degree': 2, 'gamma': 1, 'kernel': 'poly'}
0.974 (+/-0.019) for {'C': 2, 'degree': 2, 'gamma': 0.1, 'kernel': 'poly'}
0.975 (+/-0.017) for {'C': 2, 'degree': 2, 'gamma': 0.01, 'kernel': 'poly'}
0.857 (+/-0.046) for {'C': 2, 'degree': 2, 'gamma': 0.001, 'kernel': 'poly'}
0.812 (+/-0.037) for {'C': 2, 'degree': 2, 'gamma': 0.0001, 'kernel': 'poly'}
```

```
0.974 (+/-0.019) for {'C': 15, 'degree': 2, 'gamma': 1, 'kernel': 'poly'}
0.974 (+/-0.019) for {'C': 15, 'degree': 2, 'gamma': 0.1, 'kernel': 'poly'}
0.975 (+/-0.019) for {'C': 15, 'degree': 2, 'gamma': 0.01, 'kernel': 'poly'}
0.936 (+/-0.037) for {'C': 15, 'degree': 2, 'gamma': 0.001, 'kernel': 'poly'}
0.812 (+/-0.037) for {'C': 15, 'degree': 2, 'gamma': 0.0001, 'kernel': 'poly'}
0.966 (+/-0.020) for {'C': 15, 'degree': 3, 'gamma': 1, 'kernel': 'poly'}
0.966 (+/-0.020) for {'C': 15, 'degree': 3, 'gamma': 0.1, 'kernel': 'poly'}
0.967 (+/-0.020) for {'C': 15, 'degree': 3, 'gamma': 0.01, 'kernel': 'poly'}
0.829 (+/-0.024) for {'C': 15, 'degree': 3, 'gamma': 0.001, 'kernel': 'poly'}
0.715 (+/-0.279) for {'C': 15, 'degree': 3, 'gamma': 0.0001, 'kernel': 'poly'}
0.945 (+/-0.029) for {'C': 20, 'degree': 1, 'gamma': 1, 'kernel': 'poly'}
0.945 (+/-0.029) for {'C': 20, 'degree': 1, 'gamma': 0.1, 'kernel': 'poly'}
0.951 (+/-0.025) for {'C': 20, 'degree': 1, 'gamma': 0.01, 'kernel': 'poly'}
0.956 (+/-0.027) for {'C': 20, 'degree': 1, 'gamma': 0.001, 'kernel': 'poly'}
0.946 (+/-0.025) for {'C': 20, 'degree': 1, 'gamma': 0.0001, 'kernel': 'poly'}
0.974 (+/-0.019) for {'C': 20, 'degree': 2, 'gamma': 1, 'kernel': 'poly'}
0.974 (+/-0.019) for {'C': 20, 'degree': 2, 'gamma': 0.1, 'kernel': 'poly'}
0.975 (+/-0.018) for {'C': 20, 'degree': 2, 'gamma': 0.01, 'kernel': 'poly'}
0.942 (+/-0.034) for {'C': 20, 'degree': 2, 'gamma': 0.001, 'kernel': 'poly'}
0.812 (+/-0.037) for {'C': 20, 'degree': 2, 'gamma': 0.0001, 'kernel': 'poly'}
0.966 (+/-0.020) for {'C': 20, 'degree': 3, 'gamma': 1, 'kernel': 'poly'}
0.966 (+/-0.020) for {'C': 20, 'degree': 3, 'gamma': 0.1, 'kernel': 'poly'}
0.967 (+/-0.020) for {'C': 20, 'degree': 3, 'gamma': 0.01, 'kernel': 'poly'}
0.837 (+/-0.033) for {'C': 20, 'degree': 3, 'gamma': 0.001, 'kernel': 'poly'}
0.715 (+/-0.279) for {'C': 20, 'degree': 3, 'gamma': 0.0001, 'kernel': 'poly'}
```

```
0.765 (+/-0.196) for {'C': 2, 'gamma': 1, 'kernel': 'rbf'}
0.920 (+/-0.015) for {'C': 2, 'gamma': 0.1, 'kernel': 'rbf'}
0.978 (+/-0.016) for {'C': 2, 'gamma': 0.01, 'kernel': 'rbf'}
0.951 (+/-0.025) for {'C': 2, 'gamma': 0.001, 'kernel': 'rbf'}
0.915 (+/-0.042) for {'C': 2, 'gamma': 0.0001, 'kernel': 'rbf'}
0.765 (+/-0.196) for {'C': 5, 'gamma': 1, 'kernel': 'rbf'}
0.920 (+/-0.015) for {'C': 5, 'gamma': 0.1, 'kernel': 'rbf'}
0.977 (+/-0.017) for {'C': 5, 'gamma': 0.01, 'kernel': 'rbf'}
0.956 (+/-0.029) for {'C': 5, 'gamma': 0.001, 'kernel': 'rbf'}
0.934 (+/-0.032) for {'C': 5, 'gamma': 0.0001, 'kernel': 'rbf'}
0.765 (+/-0.196) for {'C': 15, 'gamma': 1, 'kernel': 'rbf'}
0.920 (+/-0.015) for {'C': 15, 'gamma': 0.1, 'kernel': 'rbf'}
0.977 (+/-0.018) for {'C': 15, 'gamma': 0.01, 'kernel': 'rbf'}
0.961 (+/-0.024) for {'C': 15, 'gamma': 0.001, 'kernel': 'rbf'}
0.948 (+/-0.024) for {'C': 15, 'gamma': 0.0001, 'kernel': 'rbf'}
0.765 (+/-0.196) for {'C': 20, 'gamma': 1, 'kernel': 'rbf'}
0.920 (+/-0.015) for {'C': 20, 'gamma': 0.1, 'kernel': 'rbf'}
0.977 (+/-0.018) for {'C': 20, 'gamma': 0.01, 'kernel': 'rbf'}
0.961 (+/-0.023) for {'C': 20, 'gamma': 0.001, 'kernel': 'rbf'}
0.951 (+/-0.024) for {'C': 20, 'gamma': 0.0001, 'kernel': 'rbf'}
```

According to the result of grid searching, the parameter for the best solution is: **0.978**

0.978 (+/-0.016) for {'C': 2, 'gamma': 0.01, 'kernel': 'rbf'}

As the result shows, linear method always has a better precision than 90%. Polynomial method has a big various since the degree changes (same as the reason we have discussed at question 1). rbf also has a big various of precision, but it also has the best precision. It is because rbf sometimes overfits (I will discuss it later in the discussion part), so error rate increases. However, if the parameters were chosen carefully, rbf is much better than the other method.

Question 3:

```
gamma = [1, 1e-1, 1e-2, 1e-3]

for gamma_ in gamma:
    tStart = time()
    print("User-Defined Starts... (for gamma=%g)" %gamma_)
1   svm = SVC(kernel='precomputed')
    k_lin = linear_kernel(X_train, X_train)
2   k_RBF = rbf_kernel(X_train, X_train, gamma_)
    gram_train = k_RBF+k_lin
3   svm.fit(gram_train, T_train)

    k_lin = linear_kernel(X_test, X_train)
    k_RBF = rbf_kernel(X_test, X_train, gamma_)
4   gram_test = k_RBF+k_lin

    error_rate = 0
    for i, v in enumerate(svm.predict(gram_test)):
        if v!= T_test[i]:
5           error_rate+=1
    error_rate = error_rate / len(X_test)
    print("Precision : {:.3%}".format(1-error_rate))

    tEnd = time()
    print("User-Defined Kernel costs : %.2f sec" %(tEnd - tStart))
    print("----------")
```

```
User-Defined Starts... (for gamma=1)
Precision : 95.640%
User-Defined Kernel costs : 2.05 sec
----------
User-Defined Starts... (for gamma=0.1)
Precision : 95.640%
User-Defined Kernel costs : 1.96 sec
----------
User-Defined Starts... (for gamma=0.01)
Precision : 95.320%
User-Defined Kernel costs : 2.15 sec
----------
User-Defined Starts... (for gamma=0.001)
Precision : 95.080%
User-Defined Kernel costs : 2.08 sec
```

1. Since it is user-defined kernel, the parameter is set as ‘precomputed’
2. Calculate the gram matrix for training data
3. Fit gram matrix and training label to the model
4. Calculate the gram matrix for testing data
5. Calculate the precision
6. Calculate new SVM model with different gamma

As the gamma decreases, the precision also decreases. However, the precision is higher than the pure rbf method when gamma=1. And also higher when gamma=0.1.

RBF

```
tStart = time()
print("RBF Starts...")

svm = SVC(kernel='rbf', gamma=1)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("RBF Kernel costs : %.2f sec" %(tEnd - tStart))
```

```
RBF Starts...
Precision : 30.040%
RBF Kernel costs : 55.37 sec
```

RBF

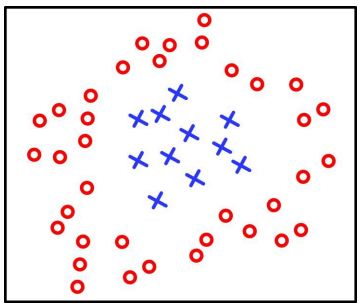
```
tStart = time()
print("RBF Starts...")

svm = SVC(kernel='rbf', gamma=0.1)
svm.fit(X_train,T_train)

error_rate = 0
for i, v in enumerate(svm.predict(X_test)):
    if v!= T_test[i]:
        error_rate+=1
error_rate = error_rate / len(X_test)
print("Precision : {:.3%}".format(1-error_rate))

tEnd = time()
print("RBF Kernel costs : %.2f sec" %(tEnd - tStart))
```

```
RBF Starts...
Precision : 90.400%
RBF Kernel costs : 49.59 sec
```

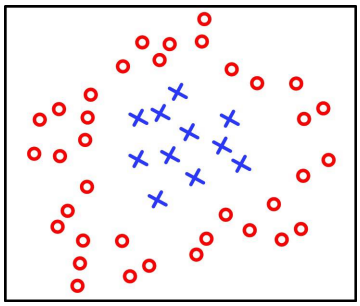
Maybe it is because linear method will prevent rbf from overfitting when it has large gamma.

Discussion

Linear

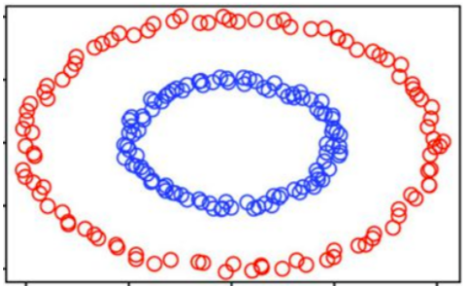
Pros	Cons
Fastest way. Not easy to overfit.	Isn't always solvable, as the graph below. 

Polynomial

Pros	Cons
More flexible than linear.	$$K(\mathbf{x}, \mathbf{x}') = (\zeta + \gamma \mathbf{x}^T \mathbf{x}')^Q$$ Difficult to calculate when Q is large. Even though it is more flexible than linear, it still can't solve the graph below. 

rbf

Pros	Cons
Much more powerful than the	Slower than linear.

above method since it can solve multi-dimension problem.  Easier to calculate than polynomial method.	Sometimes too powerful so that the solution isn't usable. (Overfit)