Homework #5

Importing the relevant libraries

In this homework, I will mainly utilise the scikit-learn package, which implements the libsvm library to handle all of the internal computations.

In [134]:
```
from sklearn import svm
from sklearn.model_selection import GridSearchCV
import pandas as pd
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics.pairwise import linear_kernel,rbf_kernel
```

Importing the data

We will use the numpy.asmatrix and pandas.read_csv methods to convert the csv files datas into numpy matrices/arrays for ease of handling.

In [35]:
```
train_data = np.asmatrix(pd.read_csv('X_train.csv'))
train_label = np.asmatrix(pd.read_csv('T_train.csv'))
test_data = np.asmatrix(pd.read_csv('X_test.csv'))
test_label = np.asmatrix(pd.read_csv('T_test.csv'))
```

Building the models and performing the predictions

As mentioned, The implementation is based on libsvm. The multiclass support is handled according to a one-vs-one scheme. We will use all the default parameters given by the API.

Code explanation:

- clf = svm.SVC(kernel) build model with selected kernel
- clf.fit(train_data, train_label) train the model with the training data
- predict = clf.predict(test_data) output the predicted labels of the test data

Linear model

In [122]:
```
clflin = svm.SVC(kernel='linear')
clflin.fit(train_data, train_label)
linear_predict=clflin.predict(test_data)
```

Polynomial model

In [68]:
```
clfpoly = svm.SVC(kernel='poly')
clfpoly.fit(train_data, train_label)
poly_predict=clfpoly.predict(test_data)
```

RBF model

In [52]:
```
clfrbf = svm.SVC(kernel='rbf')
clfrbf.fit(train_data, train_label)
rbf_predict=clfrbf.predict(test_data)
```

Evaluating the performance of each model

We will first compare the prediction against the actual label to compute the total accuracy of each kernel, i.e. correctly predicted/total.

In [123]:
```
wronglin,wrongpoly,wrongrbf=0,0,0
total=0
for i in range(len(test_label)):
    total+=1
    l=test_label[i]
    if(l!=linear_predict[i]):
        wronglin+=1
    if(l!=poly_predict[i]):
        wrongpoly+=1
    if(l!=rbf_predict[i]):
        wrongrbf+=1
print("linear kernel accuracy: " + str((1-(wronglin/total))*100) + "%")
print("polynomial kernel accuracy: " + str((1-(wrongpoly/total))*100) + "%
")
print("rbf kernel accuracy: " + str((1-(wrongrbf/total))*100) + "%")
```

```
linear kernel accuracy: 95.07803121248499%
polynomial kernel accuracy: 34.53381352541016%
rbf kernel accuracy: 95.31812725090036%
```

We will then use sklearn's in-built functions to output a more detailed analysis of each kernel's performance. This includes the Confusion Matrix, and classification report, containing the precision, recall, f1-score, support and the averages.

The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of beta. beta == 1.0 means recall and precision are equally important.

The support is the number of occurrences of each class in y_true.

In [124]:
```
print("linear kernel")
print("\n")
print(confusion_matrix(test_label,linear_predict))
print("\n")
print(classification_report(test_label,linear_predict))
print("\n")
print("polynomial kernel")
print("\n")
print(confusion_matrix(test_label,poly_predict))
print("\n")
print(classification_report(test_label,poly_predict))
print("\n")
print("rbf kernel")
print("\n")
print(confusion_matrix(test_label,rbf_predict))
print("\n")
print(classification_report(test_label,rbf_predict))
print("\n")
```

```
linear kernel


[[488   0  10   0   1]
 [  1 491   5   2   1]
 [  4  15 453  22   6]
 [  3  16  26 454   1]
 [  1   5   4   0 490]]

              precision    recall  f1-score   support

           1       0.98      0.98      0.98       499
           2       0.93      0.98      0.96       500
           3       0.91      0.91      0.91       500
           4       0.95      0.91      0.93       500
           5       0.98      0.98      0.98       500

   micro avg       0.95      0.95      0.95      2499
   macro avg       0.95      0.95      0.95      2499
weighted avg       0.95      0.95      0.95      2499



polynomial kernel


[[210 285   2   0   2]
 [  0 500   0   0   0]
 [  0 453  47   0   0]
 [  0 452   1  47   0]
 [  0 441   0   0  59]]

              precision    recall  f1-score   support

           1       1.00      0.42      0.59       499
           2       0.23      1.00      0.38       500
           3       0.94      0.09      0.17       500
           4       1.00      0.09      0.17       500
           5       0.97      0.12      0.21       500

   micro avg       0.35      0.35      0.35      2499
   macro avg       0.83      0.35      0.31      2499
weighted avg       0.83      0.35      0.30      2499



rbf kernel


[[484   0   9   1   5]
 [  0 489   6   3   2]
 [  5   9 457  16  13]
 [  0  10  25 457   8]
 [  1   2   2   0 495]]

              precision    recall  f1-score   support

           1       0.99      0.97      0.98       499
           2       0.96      0.98      0.97       500
           3       0.92      0.91      0.91       500
           4       0.96      0.91      0.94       500
```

```
          5       0.95      0.99      0.97       500

  micro avg       0.95      0.95      0.95      2499
  macro avg       0.95      0.95      0.95      2499
weighted avg       0.95      0.95      0.95      2499
```

As seen from above, linear and rbf kernel both have a high overall accuracy of 95%, while the polynomial kernel only had around 34% accuracy, so it seems as though the polynomial kernel may not be suitable for this particular dataset. From the confusion matrix, we can see that most of the error in the polynomail kernel come from the label "2", meaning that it wrongly classified many of the images as "2". This shows that the parameters may not be optimised, causing such a huge error rate. We will therefore perform the C-SVC with optimised input below.

C-SVC with optimised parameters

We will first use GridSearchCV help us find the best parameters. A search consists of:

```
an estimator (regressor or classifier such as sklearn.svm.SVC());
a parameter space;
a method for searching or sampling candidates;
a cross-validation scheme; and
a score function
```

For given values, GridSearchCV exhaustively considers all parameter combinations. By default, parameter search uses the score function of the estimator to evaluate a parameter setting. These are the sklearn.metrics.accuracy_score for classification and sklearn.metrics.r2_score for regression. For some applications, other scoring functions are better suited, but in this case, we will just use the default score function provided.

First, we will list out the available parameters to optimise with exhaustive grid search.

```
* linear: C
* rbf: C, gamma
* polynomial: C, gamma, degree, coef0
```

Due to time and computational power constraint, we are unable to test too many different options, hence I have chosen to use 3 to 4 parameters for each kernel. Due to the limited options available, we must choose our search options very carefully. After a few experiments, I have found that it is most practical and efficient to use exponentially increasing sequences as our options. With reference to the default parameter values for the above parameters, I have chosen the below options:

In [100]:
```
param_grid = [
  {'C': [0.1,1, 10, 100], 'kernel': ['linear']},
  {'C': [0.1,1, 10, 100], 'gamma': [10,1,0.1,0.001], 'kernel': ['rbf']},
  {'C': [0.01,1, 50], 'gamma': [0.01,1,50], 'degree':[3,4,5],'coef0':[0.01
,1,50],'kernel': ['poly']},
 ]
grid_search = GridSearchCV(svm.SVC(), param_grid)
grid_search.fit(train_data,train_label)
grid_search.best_params_
```

```
In [146]: print(grid_search.best_params_)
```

```
{'coef0': 50, 'degree': 4, 'kernel': 'poly', 'gamma': 1, 'C': 0.01}
```

So according to the result, the best performing model is the polynomial kernel, with parameters {'coef0': 50, 'degree': 4, 'gamma': 1, 'C': 0.01}. This is surprising as it was the worst performing kernel before optimisation. This shows how important it is to tune the hyperparameters used in the SVM's kernels. If time permits, one may also perform a second round of grid search with a closer range around the current-optimised parameter to further fine-tune the values.

Let us test our model with the optimised parameters:

In [102]:
```
clfpoly2 = svm.SVC(C=0.01, coef0 = 50, degree= 4, gamma= 1, kernel= 'poly'
)
clfpoly2.fit(train_data, train_label)
poly_predict2=clfpoly2.predict(test_data)
```

In [147]:
```
wrongpoly2=0
total=0
for i in range(len(test_label)):
    total+=1
    l=test_label[i]
    if(l!=poly_predict2[i]):
        wrongpoly2+=1
print("optimised polynomial kernel accuracy: " + str((1-(wrongpoly2/total)
)*100) + "%")
```

```
optimised polynomial kernel accuracy: 98.0392156862745%
```

In [148]:
```
print("optimised polynomial kernel")
print("\n")
print(confusion_matrix(test_label,poly_predict2))
print("\n")
print(classification_report(test_label,poly_predict2))
```

```
optimised polynomial kernel


[[494   0   4   0   1]
 [  0 492   7   0   1]
 [  1   1 489   7   2]
 [  0   7  15 478   0]
 [  0   3   0   0 497]]


             precision    recall  f1-score   support

          1       1.00      0.99      0.99       499
          2       0.98      0.98      0.98       500
          3       0.95      0.98      0.96       500
          4       0.99      0.96      0.97       500
          5       0.99      0.99      0.99       500

  micro avg       0.98      0.98      0.98      2499
  macro avg       0.98      0.98      0.98      2499
weighted avg       0.98      0.98      0.98      2499
```

Indeed as shown, the optimised polynomial kernel outperforms both the linear and the rbf kernel.

Combined kernel

We will now try to create combined-kernel models. First we will try to define a custom kernel using addition of a linear and rbf kernel

In [137]:
```
def custom(X,Y):
        linear = linear_kernel(X, Y)
        rbf = rbf_kernel(X, Y)
        return linear+rbf
clfcombined = svm.SVC(kernel=custom)
clfcombined.fit(train_data, train_label)
combined_predict=clfcombined.predict(test_data)
```

In [139]:
```
wrongcustom=0
total=0
for i in range(len(test_label)):
    total+=1
    l=test_label[i]
    if(l!=combined_predict[i]):
        wrongcustom+=1
print("combined kernel accuracy: " + str((1-(wrongcustom/total))*100) + "%
")
```

```
combined kernel accuracy: 95.07803121248499%
```

In [141]:
```
print("combined kernel")
print("\n")
print(confusion_matrix(test_label,combined_predict))
print("\n")
print(classification_report(test_label,combined_predict))
```

```
combined kernel


[[488   0  10   0   1]
 [  1 491   5   2   1]
 [  4  15 453  22   6]
 [  3  16  26 454   1]
 [  1   5   4   0 490]]


             precision    recall  f1-score   support

          1       0.98      0.98      0.98       499
          2       0.93      0.98      0.96       500
```

```
           3       0.91      0.91      0.91       500
           4       0.95      0.91      0.93       500
           5       0.98      0.98      0.98       500

   micro avg       0.95      0.95      0.95      2499
   macro avg       0.95      0.95      0.95      2499
weighted avg       0.95      0.95      0.95      2499
```

It's performance is approximately the same as the standalone linear and rbf kernel.

Let us now try something different. We shall use another custom kernel which uses the multiplication of the linear and rbf kernel instead.

In [142]:
```
def custom2(X,Y):
        linear = linear_kernel(X, Y)
        rbf = rbf_kernel(X, Y)
        return linear*rbf
clfcombined2 = svm.SVC(kernel=custom2)
clfcombined2.fit(train_data, train_label)
combined_predict2=clfcombined2.predict(test_data)
```

In [143]:
```
wrongcustom2=0
total=0
for i in range(len(test_label)):
    total+=1
    l=test_label[i]
    if(l!=combined_predict2[i]):
        wrongcustom2+=1
print("combined kernel accuracy: " + str((1-(wrongcustom2/total))*100) + "
%")
```

```
combined kernel accuracy: 97.23889555822329%
```

In [144]:
```
print("combined kernel 2")
print("\n")
print(confusion_matrix(test_label,combined_predict2))
print("\n")
print(classification_report(test_label,combined_predict2))
```

```
combined kernel 2


[[493   0   4   1   1]
 [  0 492   7   0   1]
 [  1   4 477  15   3]
 [  0  13  14 472   1]
 [  0   4   0   0 496]]
```

```
              precision    recall  f1-score   support

           1       1.00      0.99      0.99       499
           2       0.96      0.98      0.97       500
           3       0.95      0.95      0.95       500
           4       0.97      0.94      0.96       500
           5       0.99      0.99      0.99       500

   micro avg       0.97      0.97      0.97      2499
   macro avg       0.97      0.97      0.97      2499
weighted avg       0.97      0.97      0.97      2499
```

As seen, this model has a higher precision and accuracy than all the other models aside from the optimised polynomial model. I believe it will be able to perform even better should we perform a grid search to optimise its parameters too, but that shall not be in the scope of this assignment.