Coursework 2 Solution

Team: *18019783* and *21049846*

Department of Computer Science

University College London

December 2021

Abstract

This coursework includes three parts. Each section refers to answer of one question in the coursework. Appendix A, B, C includes corresponding codes for each part. All codes are written in Python.

Contents

Label	0	1	2	3	4	5	6	7	8	9
Occurrence	1553	1269	929	824	852	716	834	792	708	821

Table 1

1 Part 1: Kernel perceptron (Handwritten Digit Classification)

1.1 Introduction

In this part of the coursework, several multi-class classification methods are investigated and compared. The data set used is *zipcombo.dat*, which is also known as MNIST handwritten digit data set [1]. This data set includes in total 9298 handwritten digits from 0 to 9. Each sample contains a label and 16×16 pixel values as attributes. Examples of digits in the data set is shown in Fig.(1). The number of occurrence of each label is shown in Table (1).
We convert single label of each digit into one-hot encoded label vector of size 1×10. If the label is 5 then the corresponding element in the label vector is +1 and the rests are -1. This method is also called "one-versus-rest" classification method. More details are discussed in Section 1.2.2.

Figure 1: Digit labels in MNIST data set.

1.2 Theory

1.2.1 Perceptron

Perceptron algorithm works for binary class classification problem. If we have input $\boldsymbol{x} \in \mathbb{R}^n$, where n is the number of attributes, and true label $t \in \{-1, 1\}$, then the prediction $y(\boldsymbol{x}) = f(\boldsymbol{w}^{\boldsymbol{T}}\phi(\boldsymbol{x}))$, where $f(\cdot)$ is an activation function and $\phi(\boldsymbol{x})$ is a basis function. For the case where $f(a) = 1$ if $a \geq 0$ and $f(a) = -1$ elsewhere, we may make class decisions based on the value of $\boldsymbol{w}^{\boldsymbol{T}}\phi(\boldsymbol{x})$. If the classifier makes a correct decision, $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{\phi_n}t_n \geq 0$. The perceptron criterion is given by

$$E_p(\boldsymbol{w}) = -\sum_{n \in M} \boldsymbol{w}^{\boldsymbol{T}}\phi(\boldsymbol{x}_n)t_n,$$

where $M = \{n : \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{\phi_n}t_n < 0\}$.
The weight vector $\boldsymbol{w}$ may be updated using stochastic gradient descent (SGD) algorithm in

online learning problem. The new weight vector $\boldsymbol{w}^{(t+1)}$ is given by

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \nabla E_n(\boldsymbol{w}) = \boldsymbol{w}^{(t)} - \eta(\boldsymbol{\phi}_n t_n),$$

where η is the learning rate, and $E_n(\boldsymbol{w}) = -\boldsymbol{w}^{\boldsymbol{T}} \phi(\boldsymbol{x}_n) t_n$. If the data points in the data set is linearly separable, then perceptron SGD will converge until a optimal decision boundary is reached. In usual online perceptron, the update rule is

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \boldsymbol{\phi}_n t_n.$$

1.2.2 Two-class polynomial kernel perceptron

In this problem, basis function $\phi(\boldsymbol{x})$ is equal to $\boldsymbol{x}$, and $f(\cdot)$ is a sign function, $sgn(\cdot)$, that returns the sign of the value inside the brakets. Thus, if we initialise the weight vector by zeros, and by observation we may see that at some time t, the weight vector is given by

$$\boldsymbol{w}^{(t)} = \sum_{i=0}^{t-1} \alpha_i \boldsymbol{x}_i,$$

where in a single epoch, α_i is the true label value, and in many epochs it represents the number of times where some sample $\boldsymbol{x}_i$ is mis-classified. The predicted label is then given by

$$\begin{aligned} \hat{y}_t &= sgn(\sum_{i=0}^{t-1} \alpha_i \boldsymbol{x}_i \cdot \boldsymbol{x}_t) \\ &= sgn(\sum_{i=0}^{t-1} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)), \end{aligned}$$

where $K(\boldsymbol{x}_i, \boldsymbol{x}_t)$ is the kernel function, which stores a subset of seen training samples. We can see that as written in kernels, the primal form is replaced by dual form of prediction. The kernel chosen is the inner product of two sample vectors, which makes the feature space to higher dimensions by using higher degree polynomial kernels. In this question we choose a polynomial kernel function of the form $K_d(\boldsymbol{p}, \boldsymbol{q}) = (\boldsymbol{p} \cdot \boldsymbol{q})^d$, where $d \in \{1, ..., 7\}$ is the degree of the polynomial, and $\boldsymbol{p}$, $\boldsymbol{q}$ are two sample vectors. For the case where $d = 1$, the complexity of computing the prediction is $O(m^3 + mn)$ for the dual form, and $O(n^3 + n)$ for the primal form, where m is the number of samples and n is the number of experts. As d increases, the complexity of making prediction in primal form increases exponentially, which reveals that it is more tractable to compute in dual form when in high-dimensional feature space. The kernel method also allows the classifier to learn a non-linear decision boundary.

1.2.3 Generalisation to k-class polynomial kernel perceptron

The method we implemented for multi-class classification is called "One-versus-rest" (OvR). This method uses binary classification algorithm to solve multi-class classification problem. For example, for samples have label 9, the problem becomes a binary classification, that is, 9 is one class and the rest is another class. For each example in MNIST dataset, its label is predicted by comparing the 'confidence' belonging to different labels, where the label with the greatest

confidence will belong to class (+1) and the rest will belong to class (-1). The confidence $C^{(k)}$ of some sample $\boldsymbol{x}_t$ belongs to class k is calculated by

$$C^{(k)} = \sum_{i=1}^{t-1} \alpha_i^{(k)} K(\boldsymbol{x}_i, \boldsymbol{x}_t),$$

where $\alpha_i^{(k)}$ is the number of times a sample $\boldsymbol{x}_i$ is mis-classified into class k, and $K(\boldsymbol{x}_i, \boldsymbol{x}_t)$ is the inner product of two sample vectors to the power of d. This kernel is a measure of similarity of two samples, the more similar the two samples, the larger their inner product. The value of d helps to amplify such similarity, thus to improve the corresponding confidence. In MNIST data set, $k \in \{0, 1, ..., 9\}$, thus for each sample we may obtain a confidence vector, each element represents how confident this sample belongs to a corresponding label. In training process, all αs are initialised to zero, and $\alpha_i^{(k)}$ is increases by one while $\alpha_i^{(\hat{k})}$ is reduced by one if a sample $\boldsymbol{x}_i$ has true label k but mis-predicted as $\hat{k}$. If the label is predicted correctly, the value of the corresponding α does not change.

1.2.4 Gaussian kernel perceptron

Gaussian kernel is generally expressed as

$$K(\boldsymbol{p}, \boldsymbol{q}) = \theta_0 exp(-\theta_1 * ||\boldsymbol{p} - \boldsymbol{q}||^2), \tag{1}$$

where $\boldsymbol{p}$ and $\boldsymbol{q}$ are two sample vectors. In this question, we choose the amplification factor, θ_0, to be 1. θ_1 is called the kernel width, it scales the distance between $\boldsymbol{p}$ and $\boldsymbol{q}$. If θ_1 is large, the exponential decays faster as two sample vectors get further apart. As θ_1 turns to infinity, the Gram matrix becomes an identity matrix. In order to preserve more features and do not overfit, we choose $\theta_1 \in \{0.005, 0.010, 0.015, 0.020, 0.025, 0.030, 0.035\}$ after some trial values of θ_1 from 0.001 to 4. Within this range, we makes the best c of each run distributed similarly to the distribution of the best d.

1.2.5 One-versus-one algorithm

One-versus-one algorithm is also a binary classification method. Unlike one-versus-rest method, it has $k(k-1)/2$ classifiers, where k is the number of classes. Denoting $f_{ij}(\boldsymbol{x})$ as a binary classifier to classify the label of some sample $\boldsymbol{x}$, where i and j are two different classes in k classes, and here i denotes positive class and j denotes a negative class. Then we have $f_{ij} = -f_{ji}$, and the label is predicted by

$$f(\boldsymbol{x}) = arg\ max_{i \in k}(\sum_{j \in k, \neq i} f_{ij}(\boldsymbol{x})),$$

which means the prediction is based on the maximum number of vote on label i.

1.3 Experimental methods and Results

1.3.1 Basic results

We perform 20 runs for $d = \{1, ..., 7\}$ and $c = \{0.005, ..., 0.035\}$, each run randomly splits *zipcombo.dat* into 80% train and 20% test. For each run, the training set undergoes 20 epochs.

degree of polynomial kernel	Training set error rate(%)	Test set error rate(%)
1	5.2635 ± 0.3069	8.6452 ± 1.0026
2	0.1876 ± 0.0865	3.6371 ± 0.6118
3	0.1096 ± 0.0683	3.2554 ± 0.4260
4	0.0652 ± 0.0448	3.1882 ± 0.4586
5	0.0578 ± 0.0276	3.1183 ± 0.4057
6	0.0437 ± 0.0289	3.1317 ± 0.3797
7	0.0450 ± 0.0309	3.0887 ± 0.4721

Table 2: One-versus-rest polynomial kernel perceptron algorithm.

Gaussian kernel width	Training set error rate(%)	Test set error rate(%)
0.005	0.1123 ± 0.0710	3.2688 ± 0.4537
0.010	0.0511 ± 0.0301	2.9731 ± 3.6133
0.015	0.0410 ± 0.0263	3.0054 ± 0.3900
0.020	0.0370 ± 0.0204	3.1559 ± 0.4244
0.025	0.0202 ± 0.0274	3.2043 ± 0.4721
0.030	0.0215 ± 0.0220	3.2070 ± 0.3858
0.035	0.0101 ± 0.0130	3.4220 ± 0.4816

Table 3: One-versus-rest Gaussian kernel perceptron algorithm.

The basic results of using polynomial kernel perceptron algorithm and Gaussian kernel perceptron algorithm are shown in Table 2 and Table 3 respectively. The result for one-versus-one polynomial kernel perceptron is shown in Table 4.

1.3.2 Cross-validation

We perform 20 runs, and for each run we split the dataset, *zipcombo.bat*, into 80% training set and 20% test set. For each training set we perform 5-fold cross-validation and run 20 epochs and select the best degree of polynomial kernel d^*, or the best kernel width for the Gaussian

degree of polynomial kernel	Training set error rate(%)	Test set error rate(%)
1	1.6671 ± 0.1976	5.9247 ± 0.6547
2	0.0578 ± 0.0231	3.3226 ± 0.4798
3	0.0417 ± 0.0230	3.1747 ± 0.5306
4	0.0350 ± 0.0202	3.2419 ± 0.3541
5	0.0329 ± 0.0182	3.2769 ± 0.3723
6	0.0343 ± 0.0177	3.2527 ± 0.4342
7	0.0316 ± 0.0159	3.4620 ± 0.5090

Table 4: One-versus-one polynomial kernel perceptron algorithm.

Best degree	5	5	4	5	3
Test set error rate(%)	3.2796	3.4409	3.5484	3.2258	2.6882
Best degree	4	4	4	4	7
Test set error rate(%)	3.4946	3.6022	2.6882	3.2258	2.8495
Best degree	5	3	5	4	4
Test set error rate(%)	3.1720	2.7419	3.3871	2.3656	2.7419
Best degree	3	4	7	4	5
Test set error rate(%)	3.0108	3.7634	2.6344	3.2796	2.6344

Table 5

Best width	0.02	0.02	0.025	0.02	0.02
Test set error rate(%)	3.0108	2.7957	3.4946	3.2258	2.8495
Best width	0.025	0.015	0.035	0.02	0.02
Test set error rate(%)	3.4946	3.1183	3.0108	2.9570	3.0108
Best width	0.03	0.02	0.02	0.015	0.02
Test set error rate(%)	3.6559	3.0645	3.0645	2.3118	3.1183
Best width	0.025	0.02	0.025	0.015	0.02
Test set error rate(%)	3.0108	4.2473	3.3871	3.1720	2.9570

Table 6

kernel c^*. Then we obtain 20 best degrees and widths, and use each of them to re-train on the corresponding full training set, and record the test error on the test set. The same procedure is applied to one-versus-one polynomial kernel perceptron as well.

The results of using one-versus-rest polynomial kernel perceptron algorithm and Gaussian kernel perceptron algorithm are shown in Table 5 and Table 6 respectively. The mean and standard deviation for d^* is 4.45 ± 1.10, and for c^* is 0.0205 ± 0.0039. Both of them are approximately the median of their range, while the mean c^* is a bit large. This reveals that the value range of c we cross-validated over is reasonable. The mean and standard deviation for the test set error rate is (3.0887 ± 0.3964) % for the polynomial kernel and (3.1479 ± 0.3871) % for the Gaussian kernel. The result for one-versus-one algorithm is shown in Table 7. The mean and standard deviation for the corresponding d^* is 3.75 ± 0.97, and for the test set error rate is (3.1290 ± 0.4895) %.

1.3.3 Confusion matrix

The confusion matrix is of size (10×10), each element represents the number of times some true label (matrix row index) is mis-classified as some false label (matrix column index). The final confusion matrix is normalized over 20 runs, each run with the corresponding training/test sets split and best degree of polynomial kernel. The result is divided into two parts and shown in Table 8 and Table 9.

Best degree	3	5	5	4	3
Test set error rate(%)	2.6344	3.1183	3.4946	3.4409	2.8495
Best degree	4	3	6	4	3
Test set error rate(%)	3.3871	3.4409	2.7419	2.9570	3.1720
Best degree	4	3	5	3	4
Test set error rate(%)	3.2796	2.6882	3.6022	2.1505	2.9570
Best degree	3	4	4	2	3
Test set error rate(%)	2.1505	4.0323	3.4946	3.6559	3.3333

Table 7

	0	1	2	3	4
0	0.0 ± 0.0	0.0 ± 0.0	0.0089 ± 0.0126	0.0041 ± 0.0074	0.0046 ± 0.0105
1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0203 ± 0.0282
2	0.0164 ± 0.0190	0.0016 ± 0.0049	0.0 ± 0.0	0.0207 ± 0.0203	0.0355 ± 0.0226
3	0.0073 ± 0.0133	0.0007 ± 0.0032	0.0224 ± 0.0185	0.0 ± 0.0	0.0033 ± 0.0086
4	0.0017 ± 0.0054	0.0170 ± 0.0190	0.0161 ± 0.0193	0.0025 ± 0.0061	0.0 ± 0.0
5	0.0234 ± 0.0188	0.0007 ± 0.0032	0.0060 ± 0.0098	0.0265 ± 0.0192	0.0158 ± 0.0178
6	0.0226 ± 0.0179	0.0067 ± 0.0111	0.0078 ± 0.0104	0.0 ± 0.0	0.0132 ± 0.0127
7	0.0010 ± 0.0046	0.0053 ± 0.0107	0.0138 ± 0.0138	0.0023 ± 0.0057	0.0156 ± 0.0154
8	0.0220 ± 0.0176	0.0122 ± 0.0132	0.0158 ± 0.0189	0.0410 ± 0.0315	0.0165 ± 0.0150
9	0.0078 ± 0.0120	0.0040 ± 0.0089	0.0027 ± 0.0067	0.0092 ± 0.0119	0.0432 ± 0.0280

Table 8: Confusion matrix part I.

	5	6	7	8	9
0	0.0168 ± 0.0220	0.0121 ± 0.0151	0.0008 ± 0.0034	0.0085 ± 0.0146	0.0049 ± 0.0077
1	0.0040 ± 0.0086	0.0073 ± 0.0124	0.0074 ± 0.0094	0.0008 ± 0.0037	0.0015 ± 0.0047
2	0.0093 ± 0.0129	0.0070 ± 0.0108	0.0279 ± 0.0220	0.0049 ± 0.0078	0.0027 ± 0.0065
3	0.0699 ± 0.0337	0.0 ± 0.0	0.0095 ± 0.0103	0.0250 ± 0.0206	0.0085 ± 0.0114
4	0.0042 ± 0.0075	0.0247 ± 0.0222	0.0067 ± 0.0112	0.0 ± 0.0	0.0234 ± 0.0170
5	0.0 ± 0.0	0.0174 ± 0.0193	0.0017 ± 0.0051	0.0189 ± 0.0203	0.0122 ± 0.0157
6	0.0097 ± 0.0104	0.0 ± 0.0	0.0 ± 0.0	0.0082 ± 0.0158	0.0010 ± 0.0044
7	0.0039 ± 0.0107	0.0 ± 0.0	0.0 ± 0.0	0.0064 ± 0.0113	0.0279 ± 0.0226
8	0.0266 ± 0.0238	0.0050 ± 0.0079	0.0076 ± 0.0114	0.0 ± 0.0	0.0081 ± 0.0108
9	0.0034 ± 0.0070	0.0 ± 0.0	0.0292 ± 0.0322	0.0065 ± 0.0151	0.0 ± 0.0

Table 9: Confusion matrix part II.

1.3.4 Hardest-to-predict images

There are several criterion we may choose to determine the hardest-predict samples. For instance, we may perform 20 runs and find the confidence of each sample in different labels and sum them, then compare the confidence of the final predicted label of each sample to find the smallest five. We may also compare the difference between the two largest confidences of each sample to determine the hardest-to-predict samples. In this question, we run 20 times, for each run we split the data into 80% training set and 20% test set. We perform 5-fold cross-validation on the training set and obtain the best degree of polynomial kernel for the corresponding split. We then train the α matrix on the 80% training set using the best degree. Then we perform the test on the entire data set and record mis-classified samples. By repeating these steps we obtain the hardest-to-predict samples as shown in Fig.(2).
It is not surprising these five samples are the hardest to predict. They are 'distinctive' handwritten digits, and even the human eyes can hardly distinguish the correct labels.

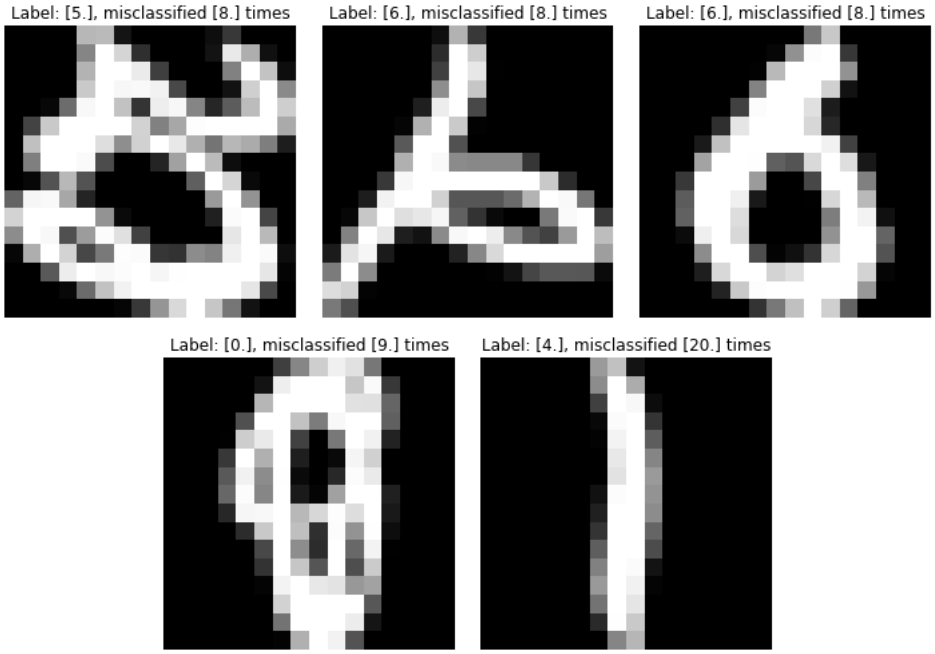

Figure 2: The five hardest-to-predict images in the data set.

1.4 Discussion

In cross validation, the parameter θ_0, as shown in Eq.(1) in Section 1.2.4 is not cross-validated over. As stated earlier, θ_0 is an amplification factor. By cross-validating over this parameter, the difference between classes is enlarged, which may help the classifier to distinguish different labels more easily. The same idea may also apply on the polynomial kernel, where an amplification factor may amplify the similarity between two samples. However, as in the polynomial kernel we take the inner product, multiplying by the same value has no effect on the classifier's ability

to distinguish different labels. A more general expression of polynomial kernel is

$$K(\boldsymbol{p}, \boldsymbol{q}) = (\boldsymbol{p} \cdot \boldsymbol{q} + \theta)^d,$$

where θ is an offset term which shifts the similarity measure by a certain amount, but it may have very limited effect on the discrimination ability of the classifier.
Although one-versus-one(OvO) algorithm requires $O(k^2)$ classifiers while one-versus-rest(OvR) requires $O(k)$ classifiers, each classifier in OvO is smaller. This makes OvO algorithm seem more efficient. If the time taken for building each classifier is approximately the same, then the efficiency of OvR may exceed that of OvO. OvR algorithm can be very effective if regularised least-square(RLS) classification algorithm applied. In RLS method, OvR algorithm can be computed very effectively by adapting matrix factorization [2]. On the other hand, OvO on support vector machine (SVM) algorithm may have faster training and obtain more accurate result if we have a large-scale data set and large number of classes [3].
The results shown in Table 2 and 3 indicate that the Gaussian kernel has relatively lower training set and test set error rate compared with the polynomial kernel. However, this depends on the chosen degree or kernel width. Different choices may result in different classifier performance. The same idea also applies to the test error obtained by the best degree or width after cross-validation.

2 Part 2: Spectral Clustering

2.1 Experiments

1. When $c = 19.698$, which is $2^{4.3}$, the two-moons data is clustered correctly with an accuracy of 100%.

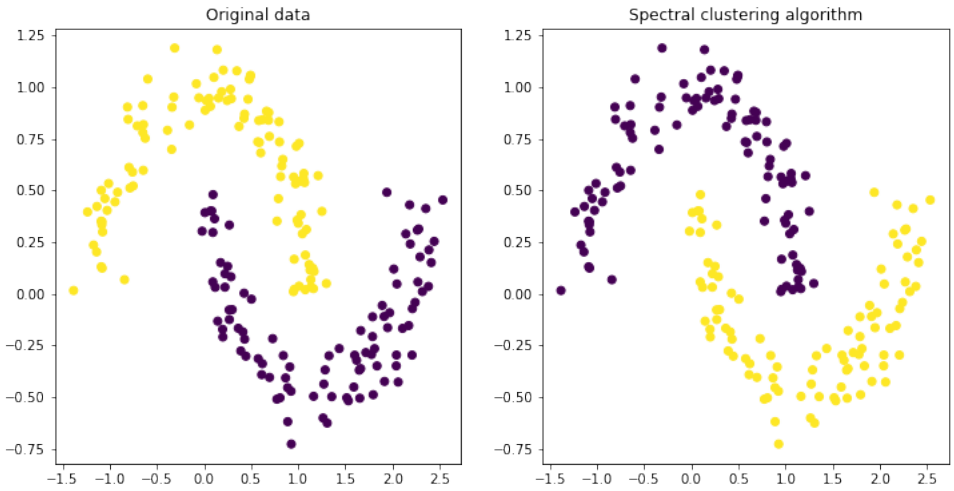

Figure 3: Two moons dataset

2. Since the gaussian data is randomly generated, the best c changes with every different generated data. For the data in this figure, when $c = 235.0$, the data is clustered correctly with an accuracy of 100%.

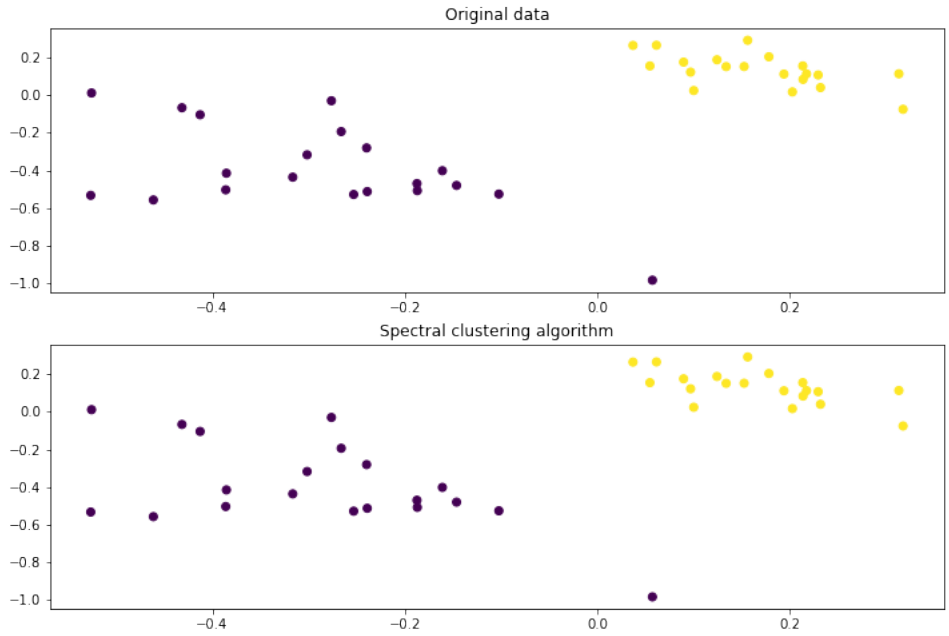

Figure 4: Gaussian dataset

3. For the data 'dtrain123', the best c is 0.01, and this c gives a correct cluster percentage (CP) of 0.91053.

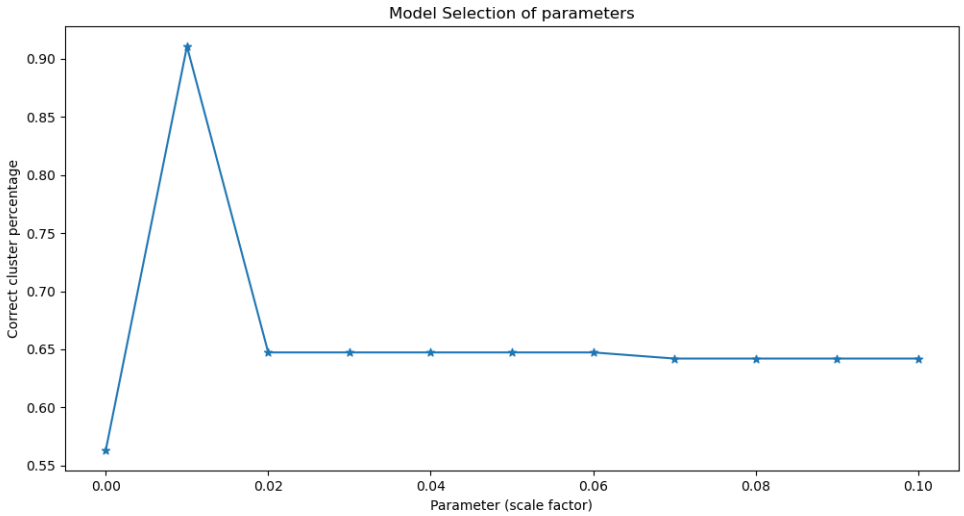

Figure 5: dtrain123 dataset

2.2 Questions

1. Explain why CP(c) is a reasonable measure of cluster correctness.

Since we know the true labels of the data, a reasonable measure of cluster correctness is the percentage of the number of correctly predicted labels. Since we only have two clusters, if we have more wrongly predicted labels than correctly predicted labels, this means we have labelled the clusters in opposite labels of the true labels, which is also correct since we only need to cluster the data into two sections. So we need to calculate the percentage of the maximum of the number of correctly predicted labels and the number of wrongly predicted labels.

2. Explain why the first eigenvalue of the Laplacian is zero and the corresponding eigenvector is the constant vector.

Since $L = diag(sum(W)) - W$, the sum of all rows of L is 0. Then the rows are not linearly independent. Hence, it is not full rank. Hence, the nullity is not zero. Then there exist $Lv = 0 = 0v$. Again, the sum of all rows of L is 0, if we have $v = (1, \cdots, 1)$, the elements of Lv is actually the sum of all rows of L, which is 0. Hence, $Lv = 0 = 0v$, L has eigenvalue 0 with corresponding eigenvector $(1, \cdots, 1)$. Since Laplacian matrix is positive semi-definite, all eigenvalue of L is greater or equal to 0. So L has the smallest eigenvalue (the first eigenvalue) as 0 and the corresponding eigenvector is the constant vector $(1, \cdots, 1)$.

3. In your own words (please explicitly cite any reference), provide a tentative explanation why spectral clustering "works" (1 paragraph only).

From [4], we can see that the spectral clustering we are doing here is actually the construction of a partition A_1, A_2 of the graph. We solve this by solving the optimization problem: $min_{A \subset V} RatioCut(A, \overline{A})$. Then we can be rewrite this using the graph's Laplacian matrix as [5]

$$f'Lf = |V| \cdot RatioCut(A, \overline{A})$$

So the optimization problem becomes $min_{A \subset V} f'Lf$, where the vector f is orthogonal to the constant one vector. Then by the Rayleigh-Ritz theorem, the solution of f is the eigenvector corresponding to the second smallest eigenvalue of L. Now, to obtain the partition, we need to transform the value of f_i by a discrete indicator function. The simplest and easiest choice is the sign function. So we use $sign(f_i)$ as the classfication label. In this way, we are able to classify data into two parts with label $(-1, +1)$.

4. Recall the parameter c in the definition (see equation (1)) of the weight matrix. Explain why and how c influences the quality of the clustering.

Here we use the Gaussian kernel as weight matrix, so c is defined as $\frac{1}{2\sigma^2}$. If c is small, the variance is large and the differences among those distance would be small and this weight matrix won't

reflect the relationships of those points well(the edges of the graph). If c is big, the variance is small and c works as an amplifier to amplify the influence of distance. However, for too large c, the influence of distance would be over amplified and give less accurate results.

3 Part 3: Sparse learning

(a) The graph below contains four plots of the sample complexity of the perceptron, winnow, least squares, and 1-nearest neighbours algorithms under dimention n. All those four plots have clear trends: linear relationship for perceptron, logarithmic relationship for winnow, linear relationship for least squares, exponential relationship for 1-nearest neighbours.

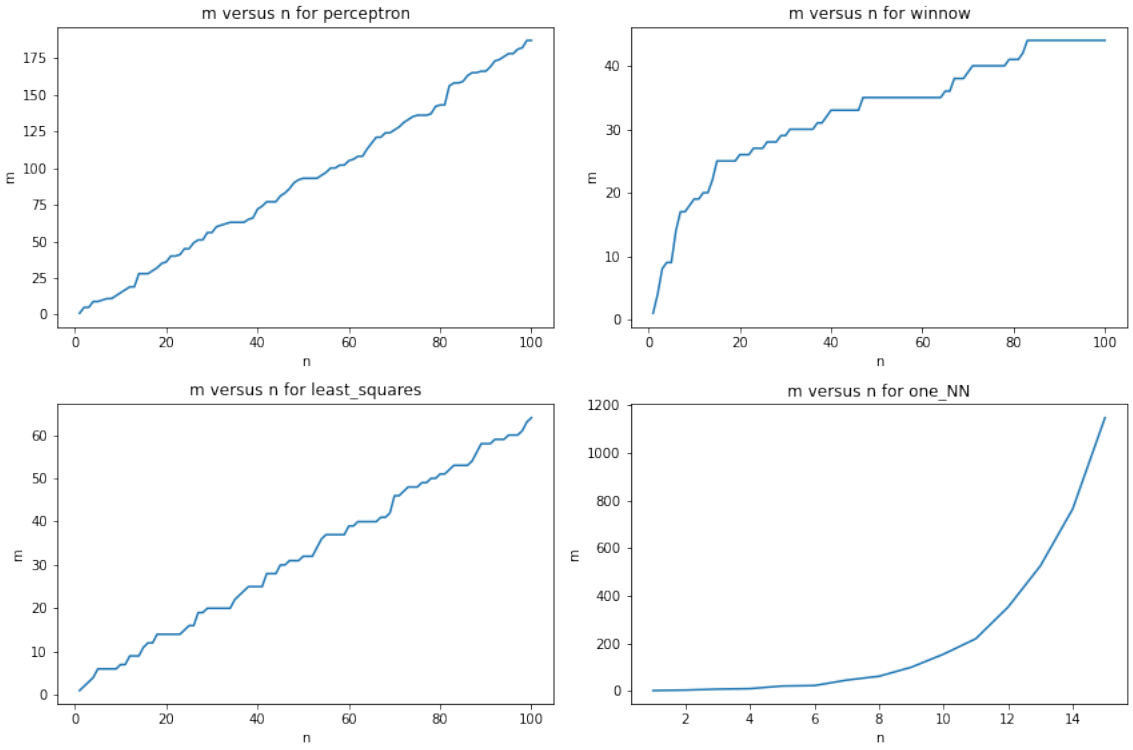

Figure 6: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for all four algorithms.

(b) i) My algorithm:

1. Run the algorithms for $1 \leq m \leq 10000$ and every n such that $1 \leq n \leq max_n$, where max_n is 15 for 1-nn and 100 for other three algorithms.

2. Choose the initial value of m for the i-th n as the final value of m for the $(i-1)$-th n. For $n = 1$, we run the algorithm from $m = 1$.

3. Generate m training samples and 5000 test samples with n features.

4. Run the algorithm for 10 times and calculate the mean generalisation error. If it is no more than 10%, move on to next n. Otherwise, try $m + 1$ and repeat step 3 and 4.

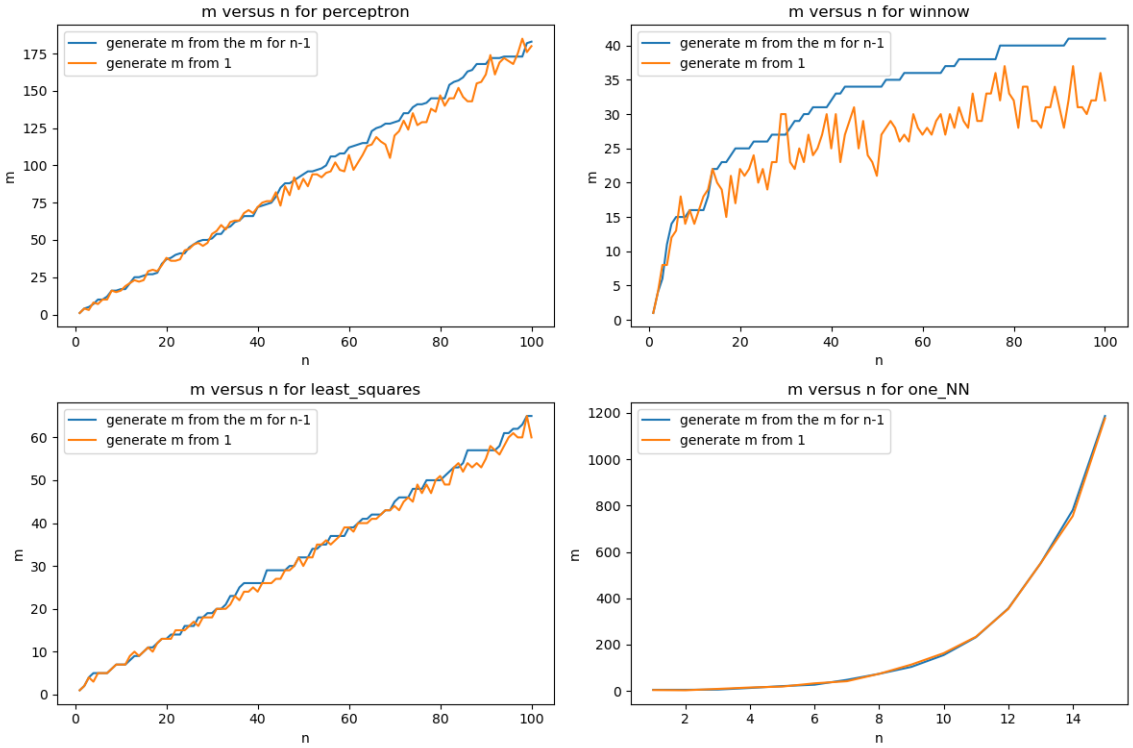

Figure 7: The comparison between my algorithm (blue line) and the normal algorithm (orange line).

ii) Trade offs and biases:

1. Sample bias: Since the test size does not cover all the combinations that can be generated, we may have biased data that will result in higher or lower generalisation error.

2. Algorithm bias: Since my algorithm assumes m always increases as n increases, if we have biased data that produces higher generalisation error, and results in a larger m for n, the final m for $n+1$ will likely be higher than the value in our normal algorithm which tests all m starting from 1. And this is why we can observe some flat line segments on the graph of my algorithm.

3. My algorithm trade-off accuracy and computation time: Since not all m has been tested for each n and there could have smaller m that fits the criteria, my accuracy is relatively lower. But the differences are not too big for our chosen n, since the intuition is m will grow as n grows. And computation time is largely reduced especially for 1-nn. For example, in the 1-nn graph, m is around 800 and 1200 for $n = 14$ and 15. In my method, we only try m from 800 until we find 1200 and save the time that we needed for $m = 1, ..., 800$. Also we will have longer computation time if we have more runs or larger test size or max_n. We know larger max_n can give us a clearer trend since we can see the relationship for more n, while bigger test size and more runs can give us a more stable trend since bias is reduced. I can't afford large max_n for 1-nn, so I chose relatively large test size and more runs to make the trend stable enough to be revealed. However, as they are not large enough, my accuracy will be lower than running through all combinations.

(c) As we stated in part(a), graphs of perceptron and least squares show linear relationships, graph of winnow shows a logarithm relationship, graph of 1-nn shows an exponential relationship. So we fit the data as those functions to estimate how m grows as a function of n and this tells us which of $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$ the algorithms belong to. And here is the graph of m versus n for all four algorithms and their fitted lines with the corresponding function form, where a, b in the graph are the parameters of the lines.

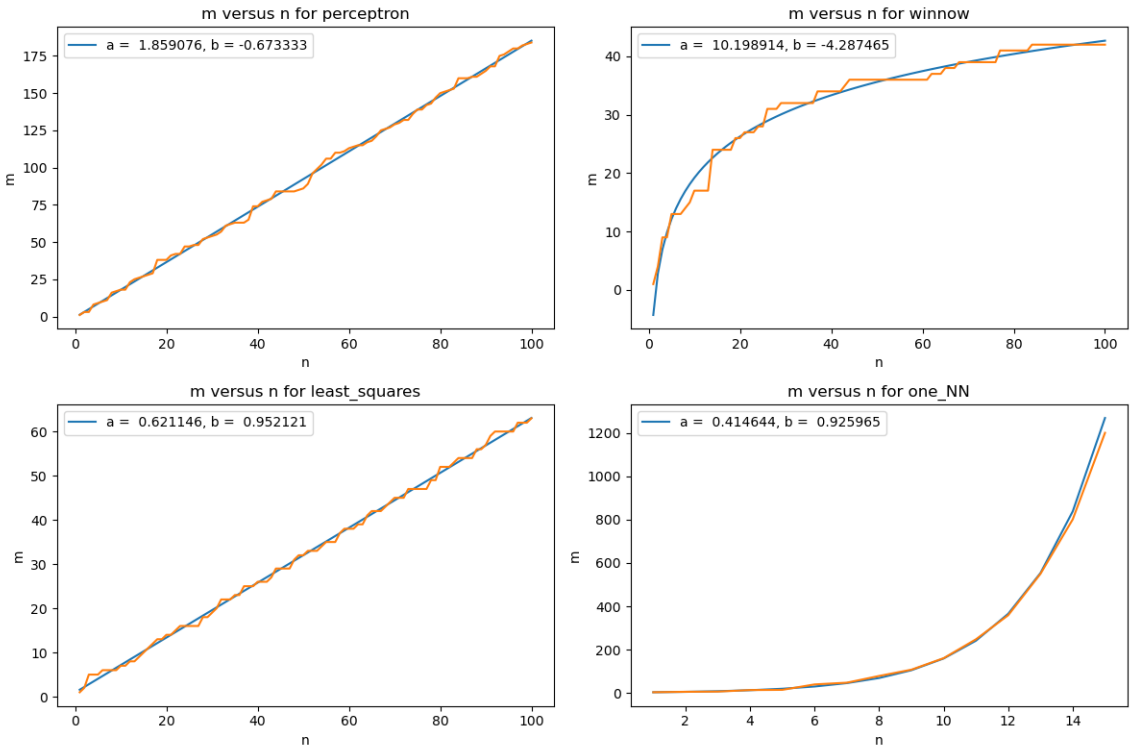

Figure 8: The plot of m and n (orange line) and the fitted line (blue line).

Here are the results we get from the graph and the actual computation time that each algorithm used when generating m and n in the graph:

Algorithm	**function form**	**fitted function**	**$O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$**	**time(seconds)**
perceptron	$a * n + b$	$1.86n - 0.67$	$\Theta(n)$	8
winnow	$a * log(n) + b$	$10.20 * log(n) - 4.29$	$\Theta(log(n))$	3
least squares	$a * n + b$	$0.62 * n + 0.95$	$\Theta(n)$	5
1-nn	$exp(b) * exp(a)^n$	$2.53 * 1.51^n$	$\Theta(1.51^n)$	2434

From the computation time, we can see while 1-nn requires a lot more time, other three algorithms only need seconds. For computational complexity (m), winnow requires the least m which is about 45 when $n = 100$ and the growth of m will be even slower as it has a logarithm relationship. Least squares are the second-best and perceptron are the third-best. They both have linear relationships but least squares requires smaller m for the same n. The most demanding one is again 1-nn, which not only requires high m, also the growth of m will be faster and faster.

It is easy to understand as adding 1 to n means adding 1 feature to each point of the data, and when calculating the distance for every pair of test and train points, we need to calculate the distance in one more dimension.

(d) Since s and the examples are sampled uniformly at random, the probability of making a mistake on example (x_s, y_s) is $\frac{M}{m}$, where M is the mistake bound for algorithm perceptron. By the Novikoff Theorem for Perceptron Bound, We have

$$M \leq (\frac{R}{\gamma})^2$$

where $R := max_t \|x_t\|$ and $\gamma \leq (\boldsymbol{v} \cdot \boldsymbol{x_t}) y_t$ and $\boldsymbol{v}$ is a vector with $\|\boldsymbol{v}\| = 1$.
Since all entries of data are sampled from -1 and $+1$, the maximized γ is 1 and $R = max_t \|x_t\| = \sqrt{n}$. As a result, the tightest bound we can have for M is

$$M \leq (\frac{\sqrt{n}}{1})^2 = \frac{n}{1} = n$$

$$So\ \hat{p}_{m,n} = \frac{M}{m} = \frac{n}{m}$$

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, p. 2278–2324, 1998.

[2] L. Rosasco, "Lecture 8- regularized least squares classification," Spring 2014.

[3] J. Milgram, M. Cheriet, and R. Sabourin, *Tenth International Workshop on Frontiers in Handwriting Recognition.*

[4] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[5] Wikipedia, "Eigenvalues and eigenvectors — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Eigenvalues%20and%20eigenvectorsoldid=1057253625, 2021. [Online; accessed 07-December-2021].

Appendices

A Part 1 code

```
# -*- coding: utf-8 -*-
"""
Created on Thu Dec  2 03:45:39 2021

@author: 21049846

0078 SL CW2
"""
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn as sns
import math
import statistics as stat
import time
from scipy.spatial.distance import cdist
from itertools import combinations

dataset = np.loadtxt('zipcombo.dat')
t = dataset[:,0].reshape((-1,1))
X = dataset[:,1:]

class_dict = {}
for i in range(int(min(t)),int(max(t)+1)):
    class_dict[str(i)] = np.count_nonzero(t==i)
num_classes = len(class_dict)

def shuffle(train, label, rs):
    """
    split zipcombo.dat into training and test set.
    """
    full_data_list = []
    full_data_list.append(train_test_split(train, label, test_size=0.2,
    random_state=rs))
    return full_data_list[0]

def polynomial_gram_matrix(X1, X2, d):
    """
    :input:
        X1: input matrix.
        X2: input matrix.
        d: degree of polynomial kernel function.
    :output:
        K: gram matrix
    """
    K = np.matmul(X1,X2.T)**d
```

```
    return K

def gaussian_gram_matrix(X1, X2, c):
    """
    :input:
        X1: input matrix.
        X2: input matrix.
        c: Gaussian kernel width.
    :output:
        K: gram matrix
    """
    K = np.exp(-c * cdist(X1,X2)**2) # problem

    return K

def test_kernel_perceptron(train_set, label, d):
    """
    :input:
        train_set: training set matrix.
        label: label vector.
        d: degree of polynomial kernel function.
    :output:
        mean and standard deviation of training set error and test set error for 20 runs.
    """
    run = 20
    epoches = 20
    k = 10
    train_error = []
    confusion = []
    test_error = []
    confusion_test = []

    for r in range(run):
        start = time.time()

        data = shuffle(train_set,label,r)

        m = data[0].shape[0]
        alpha = np.zeros((k,m))
        K = polynomial_gram_matrix(data[0],data[0],d)
        confusion_matrix = np.zeros((k,k))
        for e in range(epoches):
            num_false = 0
            for t in range(m):
                true_label = data[2][t]
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                pred_label = np.argmax(pred_y_value)
                if pred_label != true_label:
                    confusion_matrix[int(data[2][t]),int(pred_label)] += 1
```

```
                    alpha[int(pred_label),t] -= 1
                    alpha[int(true_label),t] += 1
                    num_false += 1
            train_error_rate = num_false / m

        train_error.append(train_error_rate)

        confusion.append(confusion_matrix)
        end = time.time()

        time_taken = end - start

        print("current run: {}, current train error rate: {}, time taken: {}".
    format(r+1, train_error[-1], time_taken))

        start_test = time.time()
        m_test = data[1].shape[0]
        K_test = polynomial_gram_matrix(data[0],data[1],d)
        confusion_matrix_test = np.zeros((k,k))
        num_false_test = 0

        for t in range(m_test):
            true_label = data[3][t]
            pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
            confidence_test = pred_y_value_test
            pred_label = np.argmax(confidence_test)
            if pred_label != true_label:
                num_false_test += 1
                confusion_matrix_test[int(true_label), int(pred_label)] += 1
        test_error_rate = num_false_test / m_test

        test_error.append(test_error_rate)
        confusion_test.append(confusion_matrix_test)
        end_test = time.time()

        time_taken_test = end_test - start_test
        print("current run: {}, current test error rate: {}, time taken: {}".
    format(r+1, test_error[-1], time_taken_test))

    mean_train_error = stat.mean(train_error)
    std_train_error = stat.stdev(train_error)
    avg_confusion_mat = confusion[-1] / sum(confusion)

    mean_test_error = stat.mean(test_error)
    std_test_error = stat.stdev(test_error)
    avg_confusion_matrix_test = confusion_test[-1] / sum(confusion_test)

    return [mean_train_error,std_train_error],[mean_test_error,std_test_error],
    [avg_confusion_mat,avg_confusion_matrix_test]

def k_fold_cross_val(k,data_set):
```

```
    """
    :input:
        k: number of folds
        data_set: dataset to be split into k folds
    :output:
        partitioned_set: a list contains k sub-lists, each sub-list is one fold of the dataset
    """
    if len(data_set) % k == 0:
        partitioned_set = np.reshape(data_set,(k,int(len(data_set)/k),-1))
    else:
        new_data_set = data_set.tolist()
        new_length = len(data_set) - len(data_set) % k
        left = [new_data_set[i]  for i in range(new_length, len(data_set))]
        partitioned_set = [new_data_set[i] for i in range(new_length)]
        partitioned_set = np.reshape(partitioned_set, (k, int(new_length / k), -1))
        partitioned_set = partitioned_set.tolist()
        for i in range(len(left)):
            partitioned_set[i].append(left[i])

    return partitioned_set

def cv_kernel_perceptron(train_set, label):
    """
    :input:
        train_set: training set matrix.
        label: label vector.
    :output:
        a list contains best degree of polynomial kernel and a list of corresponding test set error.
    """
    run = 20
    epoches = 20
    k = 10
    best_d = []
    test_error_best_d = []

    for r in range(run):
        start = time.time()
        data = shuffle(train_set,label,r)

        cv_data_train = k_fold_cross_val(5,data[0])
        cv_data_label = k_fold_cross_val(5,data[2])

        test_error_cv = []
        confusion_test_cv = []
        for d in range(1,8):
            test_error = []
            confusion_test = []
            for i in range(5):
```

```
                cv_test = np.array(cv_data_train[i])
                cv_test_label = np.array(cv_data_label[i])
                cv_train = []
                cv_train_label = []
                for j in range(5):
                    if j != i:
                        cv_train = cv_train + cv_data_train[j]
                        cv_train_label = cv_train_label + cv_data_label[j]
                cv_train = np.array(cv_train)
                cv_train_label = np.array(cv_train_label)

                cv_m = cv_train.shape[0]
                cv_alpha = np.zeros((k,cv_m))
                cv_K = polynomial_gram_matrix(cv_train,cv_train,d)

                for e in range(epoches):
                    for t in range(cv_m):
                        true_label = cv_train_label[t]
                        pred_y_value = np.dot(cv_alpha,cv_K[:,t].reshape((cv_m
    ,-1)))
                        confidence = pred_y_value
                        pred_label = np.argmax(confidence)

                        if pred_label != true_label:
                            cv_alpha[int(pred_label),t] -= 1
                            cv_alpha[int(true_label),t] += 1

                end = time.time()

                time_taken = end - start

                print("current run: {}, time taken: {}".format(r+1, time_taken))

                start_test = time.time()
                m_test_cv = cv_test.shape[0]
                K_test_cv = polynomial_gram_matrix(cv_train,cv_test,d)
                confusion_matrix_test_cv = np.zeros((k,k))
                num_false_test = 0

                for t in range(m_test_cv):
                    true_label = cv_test_label[t]
                    pred_y_value_test = np.dot(cv_alpha, K_test_cv[:,t].reshape
    ((cv_m,-1)))
                    confidence_test = pred_y_value_test
                    pred_label = np.argmax(confidence_test)
                    if pred_label != true_label:
                        num_false_test += 1
                        confusion_matrix_test_cv[int(true_label), int(pred_label
    )] += 1
                test_error_rate = num_false_test / m_test_cv
                test_error.append(test_error_rate)
```

```
                confusion_test.append(confusion_matrix_test_cv)
                end_test = time.time()
                time_taken_test = end_test - start_test
                print("current run: {}, d: {}, current test error rate: {}, time
    taken: {}".format(r+1, d, test_error[-1], time_taken_test))

            mean_test_error_cv = stat.mean(test_error)
            test_error_cv.append(mean_test_error_cv)
            avg_confusion_matrix_test = confusion_test[-1] / sum(confusion_test)
            confusion_test_cv.append(avg_confusion_matrix_test)

        d_star = np.argmin(test_error_cv) + 1
        best_d.append(d_star)
        start = time.time()
        data = shuffle(train_set,label,r)
        m = data[0].shape[0]
        alpha = np.zeros((k,m))
        K = polynomial_gram_matrix(data[0],data[0],d_star)
        for e in range(epoches):
            for t in range(m):
                true_label = data[2][t]
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                pred_label = np.argmax(pred_y_value)

                if pred_label != true_label:
                    alpha[int(pred_label),t] -= 1
                    alpha[int(true_label),t] += 1

        end = time.time()
        time_taken = end - start
        print("current run: {}, best d: {}, time taken: {}".format(r+1, d_star,
    time_taken))

        start_test = time.time()
        m_test = data[1].shape[0]
        K_test = polynomial_gram_matrix(data[0],data[1],d_star)
        confusion_matrix_test = np.zeros((k,k))
        num_false_test = 0

        for t in range(m_test):
            true_label = data[3][t]
            pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
            confidence_test = pred_y_value_test
            pred_label = np.argmax(confidence_test)
            if pred_label != true_label:
                num_false_test += 1
                confusion_matrix_test[int(true_label), int(pred_label)] += 1
        test_error_rate = num_false_test / m_test

        test_error_best_d.append(test_error_rate)
```

```
289         confusion_test.append(confusion_matrix_test)
290         end_test = time.time()
291 
292         time_taken_test = end_test - start_test
293         print("current run: {}, current test error rate: {}, time taken: {}".
    format(r+1, test_error_best_d[-1], time_taken_test))
294 
295     return best_d, test_error_best_d
296 
297 def confusion_kernel_perceptron(train_set, label, d, r):
298     """
299     :input:
300         train_set: training set matrix.
301         label: label vector.
302         d: degree of polynomial kernel function.
303         r: random state, corresponding to a specific split of data set.
304     :output:
305         a confusion matrix on the test set using polynomial kernel perceptron.
306     """
307     epoches = 20
308     k = 10
309     train_error = []
310     test_error = []
311     confusion_test = []
312     start = time.time()
313     data = shuffle(train_set,label,r)
314     m = data[0].shape[0]
315     alpha = np.zeros((k,m))
316     K = polynomial_gram_matrix(data[0],data[0],d)
317     for e in range(epoches):
318         num_false = 0
319         for t in range(m):
320             true_label = data[2][t]
321             pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
322             confidence = pred_y_value
323             pred_label = np.argmax(confidence)
324             #print("test")
325             if pred_label != true_label:
326                 alpha[int(pred_label),t] -= 1
327                 alpha[int(true_label),t] += 1
328                 num_false += 1
329 
330         train_error_rate = num_false / m
331 
332     train_error.append(train_error_rate)
333     end = time.time()
334     time_taken = end - start
335     print("train error rate: {}, time taken: {}".format(train_error[-1],
    time_taken))
336 
337     start_test = time.time()
```

```
    m_test = data[1].shape[0]
    K_test = polynomial_gram_matrix(data[0],data[1],d)
    confusion_matrix_test = np.zeros((k,k))
    num_false_test = 0

    for t in range(m_test):
        true_label = data[3][t]
        pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
        confidence_test = pred_y_value_test
        pred_label = np.argmax(confidence_test)
        if pred_label != true_label:
            num_false_test += 1
            confusion_matrix_test[int(true_label), int(pred_label)] += 1
    test_error_rate = num_false_test / m_test
    test_error.append(test_error_rate)
    end_test = time.time()
    time_taken_test = end_test - start_test
    print("current test error rate: {}, time taken: {}".format(test_error[-1],
    time_taken_test))

    return confusion_matrix_test

def test_gaussian_perceptron(train_set, label, c):
    """
    :input:
        train_set: training set matrix.
        label: label vector.
        c: Gaussian kernel width.
    :output:
        mean and standard deviation of training set error and test set error for
     20 runs.
    """
    run = 20
    epoches = 20
    k = 10
    train_error = []
    confusion = []
    test_error = []
    confusion_test = []

    for r in range(run):
        start = time.time()
        data = shuffle(train_set,label,r)
        m = data[0].shape[0]
        alpha = np.zeros((k,m))
        K = gaussian_gram_matrix(data[0],data[0],c)
        confusion_matrix = np.zeros((k,k))
        for e in range(epoches):
            num_false = 0
            for t in range(m):
                true_label = data[2][t]
```

```
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                pred_label = np.argmax(confidence)

                if pred_label != true_label:
                    confusion_matrix[int(data[2][t]),int(pred_label)] += 1
                    alpha[int(pred_label),t] -= 1
                    alpha[int(true_label),t] += 1
                    num_false += 1

            train_error_rate = num_false / m

        train_error.append(train_error_rate)
        confusion.append(confusion_matrix)
        end = time.time()
        time_taken = end - start
        print("current run: {}, current train error rate: {}, time taken: {}".
    format(r+1, train_error[-1], time_taken))
        #print("confidence vector is {}".format(confidence))

        start_test = time.time()
        m_test = data[1].shape[0]
        K_test = gaussian_gram_matrix(data[0],data[1],c)
        confusion_matrix_test = np.zeros((k,k))
        num_false_test = 0

        for t in range(m_test):
            true_label = data[3][t]
            pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
            confidence_test = pred_y_value_test
            pred_label = np.argmax(confidence_test)
            if pred_label != true_label:
                num_false_test += 1
                confusion_matrix_test[int(true_label), int(pred_label)] += 1
        test_error_rate = num_false_test / m_test
        test_error.append(test_error_rate)
        confusion_test.append(confusion_matrix_test)
        end_test = time.time()
        time_taken_test = end_test - start_test
        print("current run: {}, current test error rate: {}, time taken: {}".
    format(r+1, test_error[-1], time_taken_test))

    mean_train_error = stat.mean(train_error)
    std_train_error = stat.stdev(train_error)
    mean_test_error = stat.mean(test_error)
    std_test_error = stat.stdev(test_error)

    return [mean_train_error,std_train_error],[mean_test_error,std_test_error]

def cv_gaussian_perceptron(train_set, label):
    """
```

```
436     :input:
437         train_set: training set matrix.
438         label: label vector.
439     :output:
440         a list contains best Gaussian kernel width and a list of
441 corresponding test set error.
442     """
443     run = 20
444     epoches = 20
445     k = 10
446     best_c = []
447     test_error_best_c = []
448     c = [0.005,0.010,0.015,0.020,0.025,0.030,0.035]
449 
450     for r in range(run):
451 
452         data = shuffle(train_set,label,r)
453         cv_data_train = k_fold_cross_val(5,data[0])
454         cv_data_label = k_fold_cross_val(5,data[2])
455         test_error_cv = []
456         confusion_test_cv = []
457         for d in range(0,7):
458             test_error = []
459             confusion_test = []
460             for i in range(5):
461                 cv_test = np.array(cv_data_train[i])
462                 cv_test_label = np.array(cv_data_label[i])
463                 cv_train = []
464                 cv_train_label = []
465                 for j in range(5):
466                     if j != i:
467                         cv_train = cv_train + cv_data_train[j]
468                         cv_train_label = cv_train_label + cv_data_label[j]
469                 cv_train = np.array(cv_train)
470                 cv_train_label = np.array(cv_train_label)
471                 cv_m = cv_train.shape[0]
472                 cv_alpha = np.zeros((k,cv_m))
473                 cv_K = gaussian_gram_matrix(cv_train,cv_train,c[d])
474                 start = time.time()
475 
476                 for e in range(epoches):
477                     for t in range(cv_m):
478                         true_label = cv_train_label[t]
479                         pred_y_value = np.dot(cv_alpha,cv_K[:,t].reshape((cv_m
    ,-1)))
480                         confidence = pred_y_value
481                         pred_label = np.argmax(confidence)
482 
483                         if pred_label != true_label:
484                             cv_alpha[int(pred_label),t] -= 1
485                             cv_alpha[int(true_label),t] += 1
```

```
                end = time.time()
                time_taken = end - start
                print("current run: {}, time taken: {}".format(r+1, time_taken))
                start_test = time.time()
                m_test_cv = cv_test.shape[0]
                K_test_cv = gaussian_gram_matrix(cv_train,cv_test,c[d])
                confusion_matrix_test_cv = np.zeros((k,k))
                num_false_test = 0

                for t in range(m_test_cv):
                    true_label = cv_test_label[t]
                    pred_y_value_test = np.dot(cv_alpha, K_test_cv[:,t].reshape
((cv_m,-1)))
                    confidence_test = pred_y_value_test
                    pred_label = np.argmax(confidence_test)
                    if pred_label != true_label:
                        num_false_test += 1
                        confusion_matrix_test_cv[int(true_label), int(pred_label
)] += 1
                test_error_rate = num_false_test / m_test_cv
                test_error.append(test_error_rate)
                confusion_test.append(confusion_matrix_test_cv)
                end_test = time.time()
                time_taken_test = end_test - start_test
                print("current run: {}, c: {}, current test error rate: {}, time
 taken: {}".format(r+1, c[d], test_error[-1], time_taken_test))

            mean_test_error_cv = stat.mean(test_error)
            test_error_cv.append(mean_test_error_cv)
            avg_confusion_matrix_test = confusion_test[-1] / sum(confusion_test)
            confusion_test_cv.append(avg_confusion_matrix_test)

        d_star = np.argmin(test_error_cv) + 1
        c_star = c[d_star]
        best_c.append(c_star)
        start = time.time()
        data = shuffle(train_set,label,r)
        m = data[0].shape[0]
        alpha = np.zeros((k,m))
        K = gaussian_gram_matrix(data[0],data[0],c_star)

        for e in range(epoches):
            for t in range(m):
                true_label = data[2][t]
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                pred_label = np.argmax(pred_y_value)
                if pred_label != true_label:
                    alpha[int(pred_label),t] -= 1
                    alpha[int(true_label),t] += 1
```

```
534
535         end = time.time()
536         time_taken = end - start
537         print("current run: {}, best c: {}, time taken: {}".format(r+1, c_star,
    time_taken))
538         start_test = time.time()
539         m_test = data[1].shape[0]
540         K_test = gaussian_gram_matrix(data[0],data[1],c_star)
541         confusion_matrix_test = np.zeros((k,k))
542         num_false_test = 0
543
544         for t in range(m_test):
545             true_label = data[3][t]
546             pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
547             confidence_test = pred_y_value_test
548             pred_label = np.argmax(confidence_test)
549             if pred_label != true_label:
550                 num_false_test += 1
551                 confusion_matrix_test[int(true_label), int(pred_label)] += 1
552         test_error_rate = num_false_test / m_test
553
554         test_error_best_c.append(test_error_rate)
555         confusion_test.append(confusion_matrix_test)
556         end_test = time.time()
557         time_taken_test = end_test - start_test
558         print("current run: {}, current test error rate: {}, time taken: {}".
    format(r+1, test_error_best_c[-1], time_taken_test))
559
560     return best_c, test_error_best_c
561
562 def hardest_to_predict(train_set, label, d, r):
563     """
564     :input:
565         train_set: training set matrix.
566         label: label vector.
567         d: degree of polynomial kernel function.
568         r: random state , corresponding to a specific split of data set.
569     :output:
570         a list contains the number of times each sample is mis-classified.
571     """
572     epoches = 20
573     k = 10
574     start = time.time()
575     data = shuffle(train_set,label,r)
576     m = data[0].shape[0]
577     alpha = np.zeros((k,m))
578     K = polynomial_gram_matrix(data[0],data[0],d)
579
580     for e in range(epoches):
581
582         for t in range(m):
```

```
            true_label = data[2][t]
            pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
            confidence = pred_y_value
            pred_label = np.argmax(confidence)

            if pred_label != true_label:
                alpha[int(pred_label),t] -= 1
                alpha[int(true_label),t] += 1

    end = time.time()
    time_taken = end - start
    print("current r: {}, d: {}, time taken: {}".format(r+1, d, time_taken))
    start_test = time.time()
    m_whole = train_set.shape[0]
    misprediction = np.zeros((m_whole,1))
    K_test = polynomial_gram_matrix(data[0],train_set,d)

    for t in range(m_whole):
        true_label = label[t]
        pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
        confidence_test = pred_y_value_test
        pred_label = np.argmax(confidence_test)
        if pred_label != true_label:
            misprediction[t] += 1

    end_test = time.time()
    time_taken_test = end_test - start_test
    print("current r: {}, time taken: {}".format(r+1, time_taken_test))

    return misprediction

def OvO_kernel_test(train_set, label, d):
    """
    :input:
        train_set: training set matrix.
        label: label vector.
        d: degree of polynomial kernel function.
    :output:
        mean and standard deviation of training set error and test set error for
     20 runs.
    """
    run = 20
    epoches = 20
    k = 10
    classifier = list(combinations(range(k),2))
    # Hashmap
    alpha_converter = dict(zip(tuple(classifier), tuple(range(len(classifier))))
    )
    train_error = []
    test_error = []

```

```
    for r in range(run):
        start = time.time()
        data = shuffle(train_set,label,r)
        m = data[0].shape[0]
        alpha = np.zeros((int(k*(k-1)/2),m))
        K = polynomial_gram_matrix(data[0],data[0],d)

        for e in range(epoches):
            num_false = 0
            for t in range(m):
                true_label = data[2][t]
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                binary_classifier = np.sign(confidence).clip(0,None)
                classifiers = np.zeros((len(classifier)))
                for c_index, b_index in enumerate(binary_classifier):
                    classifiers[c_index] = classifier[int(c_index)][int(b_index)
    ]

                pred_label = np.bincount(classifiers.astype(int)).argmax()
                if pred_label != true_label:
                    num_false += 1

                for loc, (a,b) in enumerate(alpha_converter.keys()):
                    pred_y = classifiers[loc]

                    if true_label == a and pred_y != true_label:
                        alpha[loc,t] -= 1
                    if true_label == b and pred_y != true_label:
                        alpha[loc,t] += 1

            train_error_rate = num_false / m

        train_error.append(train_error_rate)
        end = time.time()
        time_taken = end - start
        print("current run: {}, current train error rate: {}, time taken: {}".
    format(r+1, train_error[-1], time_taken))
        start_test = time.time()
        m_test = data[1].shape[0]
        K_test = polynomial_gram_matrix(data[0],data[1],d)
        num_false_test = 0

        for t in range(m_test):
            true_label = data[3][t]
            pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
            confidence_test = pred_y_value_test
            binary_classifier_test = np.sign(confidence_test).clip(0,None)
            classifiers_test = np.zeros((len(classifier)))
            for c_index, b_index in enumerate(binary_classifier_test):
                classifiers_test[c_index] = classifier[int(c_index)][int(b_index
```

```
    )]

            pred_label_test = np.bincount(classifiers_test.astype(int)).argmax()

            if pred_label_test != true_label:
                num_false_test += 1
        test_error_rate = num_false_test / m_test
        test_error.append(test_error_rate)
        end_test = time.time()
        time_taken_test = end_test - start_test
        print("current run: {}, current test error rate: {}, time taken: {}".format(r+1, test_error[-1], time_taken_test))

    mean_train_error = stat.mean(train_error)
    std_train_error = stat.stdev(train_error)
    mean_test_error = stat.mean(test_error)
    std_test_error = stat.stdev(test_error)

    return [mean_train_error,std_train_error],[mean_test_error,std_test_error]

def cv_0v0_perceptron(train_set, label):
    """
    :input:
        train_set: training set matrix.
        label: label vector.
    :output:
        a list contains best Gaussian kernel width and a list of corresponding test set error.
    """
    run = 20
    epoches = 20
    k = 10
    classifier = list(combinations(range(k),2))
    alpha_converter = dict(zip(tuple(classifier), tuple(range(len(classifier)))))
    best_d = []
    test_error_best_d = []

    for r in range(run):
        start = time.time()
        data = shuffle(train_set,label,r)
        cv_data_train = k_fold_cross_val(5,data[0])
        cv_data_label = k_fold_cross_val(5,data[2])
        test_error_cv = []
        for d in range(1,8):
            test_error = []
            for i in range(5):
                cv_test = np.array(cv_data_train[i])
                cv_test_label = np.array(cv_data_label[i])
                cv_train = []
                cv_train_label = []
```

```
                for j in range(5):
                    if j != i:
                        cv_train = cv_train + cv_data_train[j]
                        cv_train_label = cv_train_label + cv_data_label[j]
                cv_train = np.array(cv_train)
                cv_train_label = np.array(cv_train_label)

                cv_m = cv_train.shape[0]
                cv_alpha = np.zeros((int(k*(k-1)/2),cv_m))
                cv_K = polynomial_gram_matrix(cv_train,cv_train,d)

                for e in range(epoches):
                    for t in range(cv_m):
                        true_label = cv_train_label[t]
                        pred_y_value = np.dot(cv_alpha,cv_K[:,t].reshape((cv_m
    ,-1)))
                        confidence = pred_y_value
                        binary_classifier = np.sign(confidence).clip(0,None)

                        classifiers = np.zeros((len(classifier)))
                        for c_index, b_index in enumerate(binary_classifier):
                            classifiers[c_index] = classifier[int(c_index)][int(
    b_index)]

                        pred_label = np.bincount(classifiers.astype(int)).argmax
    ()

                        for loc, (a,b) in enumerate(alpha_converter.keys()):
                            pred_y = classifiers[loc]

                            if true_label == a and pred_y != true_label:
                                cv_alpha[loc,t] -= 1
                            if true_label == b and pred_y != true_label:
                                cv_alpha[loc,t] += 1

                end = time.time()
                time_taken = end - start
                print("current run: {}, time taken: {}".format(r+1, time_taken))
                start_test = time.time()
                m_test_cv = cv_test.shape[0]
                K_test_cv = polynomial_gram_matrix(cv_train,cv_test,d)
                num_false_test = 0

                for t in range(m_test_cv):
                    true_label = cv_test_label[t]
                    pred_y_value_test = np.dot(cv_alpha, K_test_cv[:,t].reshape
    ((cv_m,-1)))
                    confidence_test = pred_y_value_test
                    binary_classifier_test = np.sign(confidence_test).clip(0,
    None)
                    classifiers_test = np.zeros((len(classifier)))
```

```
                    for c_index, b_index in enumerate(binary_classifier_test):
                        classifiers_test[c_index] = classifier[int(c_index)][int
(b_index)]

                    pred_label_test = np.bincount(classifiers_test.astype(int)).
argmax()

                    if pred_label_test != true_label:
                        num_false_test += 1
                test_error_rate = num_false_test / m_test_cv
                test_error.append(test_error_rate)
                end_test = time.time()

                time_taken_test = end_test - start_test
                print("current run: {}, d: {}, current test error rate: {}, time
 taken: {}".format(r+1, d, test_error[-1], time_taken_test))

            mean_test_error_cv = stat.mean(test_error)
            test_error_cv.append(mean_test_error_cv)

        d_star = np.argmin(test_error_cv) + 1
        best_d.append(d_star)
        start = time.time()
        data = shuffle(train_set,label,r)
        m = data[0].shape[0]
        alpha = np.zeros((int(k*(k-1)/2),m))
        K = polynomial_gram_matrix(data[0],data[0],d_star)
        for e in range(epoches):
            num_false = 0
            for t in range(m):
                true_label = data[2][t]
                pred_y_value = np.dot(alpha,K[:,t].reshape((m,-1)))
                confidence = pred_y_value
                binary_classifier = np.sign(confidence).clip(0,None)

                classifiers = np.zeros((len(classifier)))
                for c_index, b_index in enumerate(binary_classifier):
                    classifiers[c_index] = classifier[int(c_index)][int(b_index)
]
                pred_label = np.bincount(classifiers.astype(int)).argmax()
                if pred_label != true_label:
                    num_false += 1

                for loc, (a,b) in enumerate(alpha_converter.keys()):
                    pred_y = classifiers[loc]

                    if true_label == a and pred_y != true_label:
                        alpha[loc,t] -= 1
                    if true_label == b and pred_y != true_label:
                        alpha[loc,t] += 1

```

```
821         end = time.time()
822         time_taken = end - start
823         print("current run: {}, best d: {}, time taken: {}".format(r+1, d_star,
    time_taken))
824         start_test = time.time()
825         m_test = data[1].shape[0]
826         K_test = polynomial_gram_matrix(data[0],data[1],d_star)
827         num_false_test = 0
828 
829         for t in range(m_test):
830             true_label = data[3][t]
831             pred_y_value_test = np.dot(alpha, K_test[:,t].reshape((m,-1)))
832             confidence_test = pred_y_value_test
833             binary_classifier_test = np.sign(confidence_test).clip(0,None)
834             classifiers_test = np.zeros((len(classifier)))
835             for c_index, b_index in enumerate(binary_classifier_test):
836                 classifiers_test[c_index] = classifier[int(c_index)][int(b_index
    )]
837             pred_label_test = np.bincount(classifiers_test.astype(int)).argmax()
838 
839             if pred_label_test != true_label:
840                 num_false_test += 1
841 
842         test_error_rate = num_false_test / m_test
843         test_error_best_d.append(test_error_rate)
844         end_test = time.time()
845         time_taken_test = end_test - start_test
846         print("current run: {}, current test error rate: {}, time taken: {}".
    format(r+1, test_error_best_d[-1], time_taken_test))
847 
848     return best_d, test_error_best_d
849 
850 if __name__ == "__main__":
851     print("start")
852     print(class_dict)
853     fig, axes = plt.subplots(2,5)
854     axes = axes.flatten()
855     for i in range(num_classes):
856         index = np.where(t == i)
857         image = np.reshape(X[index], (16, 16))
858         axes[i].imshow(image, cmap='gray')
859         axes[i].axis('off')
860     fig.subplots_adjust(wspace=0, hspace=0)
861     plt.tight_layout()
862     plt.show()
863 
864     for i in range(1,8):
865         print("test_kernel_perceptron(X,t,{})".format(i))
866         test_kernel_perceptron(X,t,i)
867 
868     c = [0.005, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035]
```

```
    for j in range(8):
        print("test_gaussian_perceptron(X,t,{})".format(c[j]))
        test_gaussian_perceptron(X,t,c[j])


    print("cv_kernel_perceptron(X,t)")
    best_d, test_error_d = cv_kernel_perceptron(X,t)
    best_c, test_error_c = cv_gaussian_perceptron(X,t)

    mean_d, std_d = stat.mean(best_d), stat.stdev(best_d)
    mean_error_d, std_error_d = stat.mean(test_error_d), stat.stdev(test_error_d
    )
    mean_c, std_c = stat.mean(best_c), stat.stdev(best_c)
    mean_error_c, std_error_c = stat.mean(test_error_c), stat.stdev(test_error_c
    )


    # best_d = [5,5,4,5,3,4,4,4,4,7,5,3,5,4,4,3,4,7,4,5]
    # best_c = [0.02,0.02,
    0.025,0.02,0.02,0.025,0.015,0.015,0.02,0.02,0.03,0.02,0.02,0.015,0.02,0.025,0.02,
     0.025,0.015,0.02]

    confusion_matrix_list = []
    misclassified = []
    for i in range(len(best_d)):
        misclassified.append(hardest_to_predict(X,t,best_d[i],i))
        cmt = confusion_kernel_perceptron(X,t,best_d[i],i)
        norm_cmt = cmt / np.sum(cmt)
        confusion_matrix_list.append(norm_cmt)


    mean_cm = np.array(sum(confusion_matrix_list)) / len(confusion_matrix_list)
    std_cm_list = []
    for i in range(10):
        for j in range(10):
            element_list = []
            for r in range(len(best_d)):
                element_list.append(confusion_matrix_list[r][i,j])
            std = stat.stdev(element_list)
            std_cm_list.append(std)
    std_cm = np.array(std_cm_list).reshape((10,10))

    HtP_index = np.argpartition(sum(misclassified).reshape((len(sum(
    misclassified)),)),-5)[-5:]

    for i in range(len(HtP_index)):
        index = int(HtP_index[i])

        pics = np.reshape(X[index],(16,16))

        plt.imshow(pics, cmap='gray')
```

```
        plt.axis('off')
        plt.title("Label: {}, misclassified {} times".format(t[index],sum(
    misclassified)[index]))
        plt.show()

    for i in range(1,8):
        print("OvO_kernel_test(X,t,{})".format(i))
        OvO_kernel_test(X,y,i)

    ovo_d. ovo_test_error = cv_OvO_perceptron(X,t)
    mean_ovo_d, std_ovo_d = stat.mean(ovo_d), stat.stdev(ovo_d)
    mean_ovo_error, std_ovo_error = stat.mean(ovo_test_error), stat.stdev(
    ovo_test_error)
```

Listing 1: Part I

B Part 2 code

```
# import libraries
import numpy as np
from matplotlib import pyplot as plt

# Spectral Clustering Algorithm
def weight(X, c):
    # Calculate Adjacency weight matrix W.
    l,n = X.shape
    W = np.ones((l,l))
    for i in range(l):
        for j in range(l):
            W[i,j] = np.exp(-c * (np.linalg.norm(X[i]-X[j]))**2)
    return W

def laplacian(W):
    # Calculate Graph Laplacian L.
    l,l = W.shape
    D = np.zeros((l,l))
    for i in range(l):
        D[i,i] = np.sum(W[i])
    L = D - W
    return L

def sec_eig(L):
    # Find the second smallest eigenvector of L.
    val, vec = np.linalg.eig(L)
    a = np.argsort(val)[1]
    return vec[:,a]

def sign(x):
    # Return the sign of x.
    if x >= 0:
```

```
        y = 1
    else:
        y = -1
    return y

def spectral_cluster(X, c):
    # Use data X and c to give weight W, laplacian L,
    # cluster vector v2, and use them to do the clustering.
    W = weight(X, c)
    L = laplacian(W)
    v2 = sec_eig(L)
    y = np.ones(v2.shape)
    for i in range(len(v2)):
        y[i] = sign(v2[i])
    return y

def best_cluster(X, Y, c):
    # Implement spectral_cluster function with different c
    # to determine the best c which gives the best cluster.
    # The return value is the best c, corresponding correctness and prediction.
    l,n = X.shape
    final_y = []
    final_c = 0
    final_cor = 0
    for i in c:
        y = spectral_cluster(X, i)
        cor = max((y == Y).sum(), (-y == Y).sum())
        if final_cor < cor:
            final_cor = cor
            final_c = i
            final_y = y
    final_cor_rate = final_cor / l

    return final_c, final_cor_rate, final_y
```

Listing 2: Spectral Clustering Algorithm

```
# Load data
twomoons = np.loadtxt('C:/Users/james007/Desktop/twomoons.dat.txt')
X_moons = twomoons[:, 1:]
Y_moons = twomoons[:, 0]

# Give a sequence of c and find the best cluster of data
c1 = np.linspace(0, 100, 101)[1:]
moons_c, moons_cor_rate, moons_y = best_cluster(X_moons, Y_moons, c1)
print(f'The best c is {moons_c} and it gives a correctness of{moons_cor_rate:
    .2%}')

# Plot of the comparision
plt.figure(figsize=(12,6))

# Plot of original data
```

```
plt.subplot(121)
plt.scatter(X_moons[:, 0], X_moons[:, 1], c = Y_moons)
plt.title('Original data')

# Plot of spectral clustering
plt.subplot(122)
plt.scatter(X_moons[:, 0], X_moons[:, 1], c = moons_y)
plt.title('Spectral clustering algorithm')

plt.savefig('P2E1')
```

Listing 3: Experiment 1

```
I = np.eye(2)

# Generate random class '-1'
x_1 = np.random.multivariate_normal((-0.3,-0.3), 0.04*I, 20)
y_1 = -1 * np.ones(20)

# Generate random class '+1'
x_2 = np.random.multivariate_normal((0.15,0.15), 0.01*I, 20)
y_2 = np.ones(20)

# Combine two classes to give the data and label
X_random = np.array(list(x_1) + list(x_2))
Y_random = np.array(list(y_1) + list(y_2))

# Give a sequence of c and find the best cluster of data
c2 = np.linspace(0, 2000, 2001)[1:]
random_c, random_cor_rate, random_y = best_cluster(X_random, Y_random, c2)
print(f'The best c is {random_c} and it gives a correctness of{random_cor_rate:
    .2%}')

# Plot of the comparision
plt.figure(figsize=(12,8))

# Plot of original data
plt.subplot(211)
plt.scatter(X_random[:, 0], X_random[:, 1], c = Y_random)
plt.title('Original data')

# Plot of spectral clustering
plt.subplot(212)
plt.scatter(X_random[:, 0], X_random[:, 1], c = random_y)
plt.title('Spectral clustering algorithm')

plt.savefig('P2E2')
```

Listing 4: Experiment 2

```
# Load data
dtrain123 = np.loadtxt('C:/Users/james007/Desktop/dtrain123.dat.txt')
```

```
3
4  # Define dtrain13 to only contain data with label (1,3)
5  dtrain13 = []
6  for i in range(dtrain123.shape[0]):
7      if dtrain123[i, 0] == 1 or dtrain123[i, 0] == 3:
8          dtrain13.append(dtrain123[i])
9  dtrain13 = np.array(dtrain13)
10
11 X_dt = dtrain13[:, 1:]
12 Y_dt = dtrain13[:, 0]
13
14 #Relabel the data, change label (1,3) to label (1,-1)
15 classy = np.ones(Y_dt.shape)
16 for i in range(len(Y_dt)):
17     classy[i] = 2 * (Y_dt[i] == 1) -1
18
19 # Give a sequence of c and find the best cluster of data
20 # Give the result of best c and best CP
21 c3 = np.linspace(0, 0.1, 11)
22 CP = np.ones(len(c3))
23 for i in range(len(c3)):
24     y = spectral_cluster(X_dt, c3[i])
25     cor = max((y == classy).sum(), (-y == classy).sum())
26     CP[i] = cor / len(classy)
27 best_CP = np.max(CP)
28 best_c = c3[np.argmax(CP)]
29 print(f'The best c is {best_c} and the corresponding CP is{best_CP: .5f}')
30
31 # Plot of correctness vs c
32 plt.figure(figsize=(12,6))
33 plt.plot(c3, CP)
34 plt.xlabel('Parameter (scale factor)')
35 plt.ylabel('Correct cluster percentage')
36 plt.title('Model Selection of parameters')
37 plt.scatter(c3, CP, marker='*')
38
39 plt.savefig('P2E3.png')
```

Listing 5: Experiment 3

C Part 3 code

```
1  # Import libraries
2
3  import numpy as np
4  from matplotlib import pyplot as plt
5  from matplotlib import gridspec as gridspec
6  from datetime import datetime
7
8  # Data Generation function
```

```
9
10 def sample_01(m,n):
11     # Generate samples from 0 and 1
12     X = np.random.choice([0,1], (m,n))
13     Y = X[:, 0]
14     return X, Y
15
16 def sample_11(m,n):
17     # Generate samples from -1 and 1
18     X = np.random.choice([-1,1], (m,n))
19     Y = X[:, 0]
20     return X, Y
21
22 # Four algorithms
23
24 def perceptron(m, n):
25     # Perceptron algorithm: We generate the data and predict by w@x.
26     # Then check if the result is correct and update correspondingly.
27     X_train, Y_train = sample_11(m, n)
28     X_test, Y_test = sample_11(test_size, n)
29     W = np.zeros(n)
30     Y_pred = np.zeros(m)
31     for t in range(m):
32         Y_pred[t] = np.sign(W @ X_train[t])
33         if Y_pred[t] * Y_train[t] <= 0:
34             W += Y_train[t] * X_train[t]
35
36     # Use the final W to predict and calculate the error.
37     Y_res = np.sign(X_test @ W)
38     error = np.count_nonzero(Y_res != Y_test)
39     return Y_res, error
40
41 def winnow(m, n):
42     # Perceptron algorithm: We generate the data and predict by checking whether
       w@x is less than n.
43     # Then check if the result is correct and update correspondingly.
44     X_train, Y_train = sample_01(m, n)
45     X_test, Y_test = sample_01(test_size, n)
46     W = np.ones(n)
47     Y_pred = np.ones(m)
48     for t in range(m):
49         if W @ X_train[t] < n:
50             Y_pred[t] = 0
51         if Y_pred[t] != Y_train[t]:
52             W *= np.float_power(2, (Y_train[t] - Y_pred[t]) * X_train[t])
53
54     # Use the final W to predict and calculate the error.
55     Y_res = np.where(X_test @ W < n, 0, 1)
56     error = np.count_nonzero(Y_res != Y_test)
57     return Y_res, error
58
```

```python
def least_squares(m, n):
    # Least squares algorithm: We generate the data and predict by sign(x@w).
    # Then calculate the error.
    X_train, Y_train = sample_11(m, n)
    X_test, Y_test = sample_11(test_size, n)
    W = np.linalg.pinv(X_train) @ Y_train
    Y_res = np.sign(X_test @ W)
    error = np.count_nonzero(Y_res != Y_test)
    return Y_res, error

def one_NN(m, n):
    # One nearest neighbor algorithm: We generate the data and calculate the distance
    # between every pair of train and test point, classify the test point as the nearest train point label.
    X_train, Y_train = sample_11(m, n)
    X_test, Y_test = sample_11(test_size, n)
    Y_res = np.zeros(test_size)
    for i in range(test_size):
        distance = np.linalg.norm(X_train - X_test[i], ord = 2, axis = 1)
        Y_res[i] = Y_train[np.argmin(distance)]
    error = np.count_nonzero(Y_res != Y_test)
    return Y_res, error

# My method of finding sample complexity.
def complexity(algorithm, max_n):
    ''' My algorithm: try different m for each n no more than max_n for 10 runs
        and calculate the mean error to see whether it is <= 0.1.
        Use the final m for n as the starting m for n+1.

        Input:
            algorithm(string): name of the algorithm.
            max_n(int): the maximum of n.

        Return:
            res(list): the final m for each n no more than max_n.
    '''
    res = np.zeros(max_n)
    m = 1
    for n in range(max_n):
        while m < 10000:
            error = []
            for i in range(runs):
                error.append(algorithm(m, n+1)[1])
            if np.mean(error) / test_size <= 0.1:
                res[n] = m
                break
            m += 1
    return res

# The normal method of finding sample complexity.
```

```
def complex_m(algorithm, max_n):
    ''' Normal algorithm: try different m for each n no more than max_n for 10 runs
        and calculate the mean error to see whether it is <= 0.1.
        Use 1 as the starting m for each n.

        Input:
            algorithm(string): name of the algorithm.
            max_n(int): the maximum of n.

        Return:
            res(list): the final m for each n no more than max_n.
    '''
    res = np.zeros(max_n)
    for n in range(max_n):
        m = 1
        while m < 10000:
            error = []
            for i in range(runs):
                error.append(algorithm(m, n+1)[1])
            if np.mean(error) / test_size <= 0.1:
                res[n] = m
                break
            m += 1
    return res
```

Listing 6: Four Algorithms and two method functions

```
# Plot functions

def plot_complex(algorithm, max_n):
    ''' Plot the graph of the called algorithm.

        Input:
            algorithm(string): name of the algorithm.
            max_n(int): the maximum of n.
    '''
    sequence_n = [i for i in range(1, max_n+1) ]
    res = complexity(algorithm, max_n)
    plt.plot(sequence_n, res)
    plt.xlabel('Dimension (n)')
    plt.ylabel('Estimated number of samples (m)')
    plt.title(f'm to obtain 10% generalisation error versus n for {algorithm.__name__}.')
    plt.show()

def plot_all(algorithm_seq, max_n_seq):
    ''' Plot the graphs of the all algorithms contained in the algorithm_seq.
        And use the corresponding max_n in the max_n_seq.

        Input:
            algorithm_seq(list): list of names of the algorithms.
```

```
            max_n_seq(list): list of all maximum value of n for each algorithm.
    '''
    fig = plt.figure(figsize=(12,8))
    gs = gridspec.GridSpec(2, 2)
    ax = [algorithm_seq[k].__name__ for k in range(4)]
    ax[0] = plt.subplot(gs[0,0])
    ax[1] = plt.subplot(gs[0,1])
    ax[2] = plt.subplot(gs[1,0])
    ax[3] = plt.subplot(gs[1,1])

    for k in range(4):
        algorithm = algorithm_seq[k]
        res = complexity(algorithm, max_n_seq[k])
        sequence_n = np.array([i for i in range(1, max_n_seq[k]+1)])

        ax[k].plot(sequence_n, res)
        ax[k].set(xlabel = 'n', ylabel = 'm')
        ax[k].set_title(f'm versus n for {algorithm.__name__}')

    plt.tight_layout()
    fig.savefig('algorithm15.png')

def plot_fit(algorithm_seq, max_n_seq):
    ''' Plot the graphs of the all algorithms contained in the algorithm_seq.
        And use the corresponding max_n in the max_n_seq.
        Also plot the fitted line for each graph using the specified function
    form.

        Input:
            algorithm_seq(list): list of names of the algorithms.
            max_n_seq(list): list of all maximum value of n for each algorithm.

        Return:
            Duration(list): the computation time for each algorithm.
    '''
    fig = plt.figure(figsize=(12,8))
    gs = gridspec.GridSpec(2, 2)
    ax = [algorithm_seq[k].__name__ for k in range(4)]
    ax[0] = plt.subplot(gs[0,0])
    ax[1] = plt.subplot(gs[0,1])
    ax[2] = plt.subplot(gs[1,0])
    ax[3] = plt.subplot(gs[1,1])

    Duration = []
    for k in range(4):
        algorithm = algorithm_seq[k]
        sequence_n = np.array([i for i in range(1, max_n_seq[k]+1)])

        start_time = datetime.now()
        res = complexity(algorithm, max_n_seq[k])
        end_time = datetime.now()
```

```
        Duration.append(end_time - start_time)

        if k == 1:
            a, b = np.polyfit(np.log(sequence_n), res, 1)
            y = a * np.log(sequence_n) + b
        elif k == 3:
            a, b = np.polyfit(sequence_n, np.log(res), 1)
            y = np.exp(a * sequence_n + b)
        else:
            a, b = np.polyfit(sequence_n, res, 1)
            y = a * sequence_n + b

        ax[k].plot(sequence_n, y, label = f'a = {a: 2f}, b = {b: 2f}')
        ax[k].plot(sequence_n, res)
        ax[k].set(xlabel = 'n', ylabel = 'm')
        ax[k].set_title(f'm versus n for {algorithm.__name__}')
        ax[k].legend()

    plt.tight_layout()
    fig.savefig('algorithm15 fit.png')
    return Duration

def plot_comparison(algorithm_seq, max_n_seq):
    ''' Plot the graphs of the all algorithms contained in the algorithm_seq.
        And use the corresponding max_n in the max_n_seq.
        Use both my method and the normal method to produce m and see the
    comparison in graphs.

        Input:
            algorithm_seq(list): list of names of the algorithms.
            max_n_seq(list): list of all maximum value of n for each algorithm.

        Return:
            Duration1(list): the computation time for each algorithm using my
    method.
            Duration2(list): the computation time for each algorithm using the
    normal method.
    '''
    fig = plt.figure(figsize=(12,8))
    gs = gridspec.GridSpec(2, 2)
    ax = [algorithm_seq[k].__name__ for k in range(4)]
    ax[0] = plt.subplot(gs[0,0])
    ax[1] = plt.subplot(gs[0,1])
    ax[2] = plt.subplot(gs[1,0])
    ax[3] = plt.subplot(gs[1,1])

    Duration1 = []
    Duration2 = []
    for k in range(4):
        algorithm = algorithm_seq[k]
        sequence_n = np.array([i for i in range(1, max_n_seq[k]+1)])
```

```
122
123         start_time = datetime.now()
124         res1 = complexity(algorithm, max_n_seq[k])
125         end_time = datetime.now()
126         Duration1.append(end_time - start_time)
127
128         start_time = datetime.now()
129         res2 = complex_m(algorithm, max_n_seq[k])
130         end_time = datetime.now()
131         Duration2.append(end_time - start_time)
132
133
134         ax[k].plot(sequence_n, res1, label = 'generate m from the m for n-1')
135         ax[k].plot(sequence_n, res2, label = 'generate m from 1')
136         ax[k].set(xlabel = 'n', ylabel = 'm')
137         ax[k].set_title(f'm versus n for {algorithm.__name__}')
138         ax[k].legend()
139
140     plt.tight_layout()
141     fig.savefig('algorithm15 comparison.png')
142     return Duration1, Duration2
143
144 # Specify the parameters not included in the algorithm.
145 runs = 10
146 test_size = 5000
147 max_n_seq = [100, 100, 100, 15]
148 algorithm_seq = [perceptron, winnow, least_squares, one_NN]
149
150 # Run the algorithms and plot
151 plot_all(algorithm_seq, max_n_seq)
152 plot_fit(algorithm_seq, max_n_seq)
153 plot_comparison(algorithm_seq, max_n_seq)
```

Listing 7: Plot functions and implementation