COMP0078 Supervised Learning

Course Work 2

Number: 21069508
Number: 21136404

University College London

Academic Year: 2021/2022

Contents

1 Part 1

In this part we will explore kernel perceptron with a handwritten digit classification task. Following the experiments we divide the part into four sections. In Section 1.1, we introduce basic concepts for our implementation structure including perceptron, polynomial kernel, and OvR method. In Section 1.2 we implement four basic experiments and analyze results. Then in Section 1.3 we repeat these experiments with replacing the polynomial kernel to Gaussian kernel and compare difference. Finally, in Section 1.4 we replace OvR with OvO for further analysis.

1.1 Implementation Structure

We first introduce perceptron algorithm with polynomial kernel for a binary classification task. Then we equip the algorithm with One versus Rest (OvR) method, to generalize it to a multiple classification task. The generalized algorithm could be used as a basic structure for the handwritten task.

1.1.1 Perceptron with Polynomial Kernel

Basic Perceptron Before combining with kernel, we will introduce basic perceptron to provide a view of this binary classifier. Perceptron is a simple linear classifier with heuristic as follows:

$$\hat{\mathbf{y}}_i = \begin{cases} +1, \text{ if } \mathbf{w}^T\mathbf{x}_i > 0 \\ -1, \text{ if } \mathbf{w}^T\mathbf{x}_i \leq 0 \end{cases}$$

It can be seen from the heuristic that perceptron is aiming to find a hyperplane that can separate the data points with positive labels as well as negative labels. The only parameter that control the angle of the hyperplane is the weight $\mathbf{w}$. Weight $\mathbf{w}$ is initialized as a zero vector. During the training, weights are updated when perceptron makes a mistake on the specific data point, which can be interpreted as, the vector of hyperplane represented by weights are moving away from the mis-classified data points. Thus, the mis-classified data point will be separated apart from the previous defined area. We show a detailed algorithm below in Algorithm 1:

Definition of convergence When we transform Algorithm 1 to detailed implementation, one question is that, when will the perceptron converge to finish training? There are two situations which we need to define the convergence in the experiment. First, the data is separable. Given such data, **we say that perceptron will only be convergent when there is no mistakes detected during the training process**. In other words, only when we find a hyperplane that can completely separate the data set can we say that perceptron is converged. Second, the data is non-separable. **Then, we define the meaning of convergence as follows: During any number of the epoch of training stage, when we find that the number of mistakes detected in current epoch is larger than the last one, we say that our perceptron is approximately in the convergence stage.** The intuition behind this definition is that, we are assuming the hyperplane is moving towards to optimal angle monotonically at the beginning stage, which can be also seen as the stage of underfit. When the hyperplane is with an angle that makes more mistakes than the previous epochs, we assume that the hyperplane is "running over" the optimal angle, which causes more mistakes. Thus, the optimal angle is lying between the last epochs and the current epochs. We choose the weights of the last epoch as our convergent weights in our experiment.

Algorithm 1 Polynomial Perceptron

Input: Initialize $\alpha = \mathbf{0}$

Output: test mistakes

Data: $X_{train} \in \mathbb{R}^{m_{train} * n}$, $X_{test} \in \mathbb{R}^{m_{train} * n}$, $y_{train} \in \{-1, +1\}^{m_{train}}$, $y_{test} \in \{-1, +1\}^{m_{test}}$, train kernel matrix $K_{train} \in \mathbb{R}^{m_{train} * m_{train}}$, test kernel matrix $K_{test} \in \mathbb{R}^{m_{test} * m_{test}}$

```
/* start training                                                              */
while True do
    for i in length(X_train) do
        Prediction: y_pred = K_train[i] * α      // here * represents dot product, y_pred ∈ R^{m_train}
        if sign(y_pred) ≠ y_train[i] then
            α[i] = α[i] − sign(y_pred)           // update the weight of i-th row of K_train
        else
            α[i] = α[i]                          // weight of i-th row of K_train stay the same
    if converge then
        break                                    // Definition of convergence can be found in section 1.2.1
    else
        epoch + 1                                // restart from the first instance, running another epoch
/* start testing                                                               */
test mistakes = 0
 for i in length(X_test) do
    y_pred = K_test[i] * α
     if sign(y_pred) ≠ y_test[i] then
        test mistakes + 1
return test mistakes
```

Perceptron with Polynomial Kernel Predictions are made based on $\hat{\mathbf{y}}_i = sign(\mathbf{w}^T \mathbf{x}_i) = sign(\sum_{j=0}^{m} \alpha_j \mathbf{x}_j \mathbf{x}_i)$, thus we can use kernel to replace this inner product in order to project our data to higher dimensional space, where the classification task is easier to implement for perceptron. In our scenario, we implement Polynomial Kernel first, which is expressed as: $K_d(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \mathbf{x}_j)^d$, where d represents the number of polynomial included in the kernel expression. When we apply the polynomial kernel into perceptron algorithm, we don't strictly follow the online setting that data points are coming sequentially. They are all seen at the beginning. In original algorithm, the summation is applied to t^{th} instance, where t represents the number of sequence of incoming data: $\hat{\mathbf{y}}_t = sign(\sum_{i=0}^{t-1} \alpha K(\mathbf{x}_i, \mathbf{x}_t))$, while here we can change $t-1$ to n: $\hat{\mathbf{y}}_t = sign(\sum_{i=0}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$, where n is the number of data points in training set. Since α is initialized with all zeros, we can still satisfy the online setting for first epoch.

The motivation behind the adjustion is that, if we are strictly following the online setting, then we have to apply loop in our training process, which is relatively slow. By converting to "fake online setting", we can use vector multiplication instead of loop to implement, thus accelerating the whole process.

Similar implementation is applied to Kernel computation, instead of computing the kernel value inside loop, we can compute the Gram matrix of training set and testing set in advance. Gram matrix can be described as $G_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$. Thus we can merely ask for i^{th} column of

pre-calculated Gram matrix to complete our prediction, which is expressed as:

$$\hat{\mathbf{y}}_t = sign(\sum_{i=0}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$$

Note that when we are in the training stage, we should calculate the Gram matrix of training data set with itself, while when we are in the testing stage, $\mathbf{x}_i$ and $\mathbf{x}_j$ should come from training data set, while when we are testing, $\mathbf{x}_i$ should come from testing data set and $\mathbf{x}_j$ should come from training data set. Hence, we finish the implementation structure of perceptron algorithm with polynomial kernel.

1.1.2 One Versus Rest method

The previous structure could only solve binary classification tasks, while it cannot deal with multiple labels. For example, in our data set, we have 10 labels to identify, thus binary classifier is not enough. Here we introduce a methodology called "One Versus Rest (OvR)", which can generalize binary classifier to K-classes tasks with some trick during implementation.

The core idea of OvR is that, for a K-classes task, we can have K binary classifiers where each of them is responsible for detecting label k out of others. For each classifier $C_i, \forall i \in \{1, \cdots, k\}$, we modify the label $y_j, \forall j \in \{1, \cdots, m\}$ to be $+1$ if $y_j = i$, otherwise, the y_j is set to be -1. In our experiment based on hand-written digits data set, we have 10 classifiers from label $0, \cdots, 9$. This can be seen as the ensemble method of bagging, with each perceptron viewed as an expert. The details are shown in Algorithm 2:

Algorithm 2 One Versus Rest

Input: k Classifiers, $C_1, C_2, ..., C_k$
Output: Predicted label $\hat{y}$
Data: vector x, true label $y \in \{0, \cdots, 9\}$
max value = $-\infty$, max index = -1
for *i in range(k)* **do**
  if $i == y$ **then**
    $y_{current} = 1$ `// *change y based on binary setting`
  else
    $y_{current}$ = -1
  if $sign(C_i(x)) \neq y_{current}$ **then**
    update C_i using instance x `// *wrong prediction, update classifier i`
  else
    `/* classifier i believe instance has label i, record the confidence value */`
    max value = $sign(C_i(x))$ `/* record the index that has the maximum value */`
    max index = i
$\hat{y}$ = max index `// * assign index that has maximum confidence to be the predicted label`
return $\hat{y}$

We could see that, whenever a new instance x with true label y comes in, this instance will be predicted by 10 classifiers in our setting. If we call classifier corresponding to classify label i as C_i, then every C_i will view label i as 1 and other labels with -1. For $C_i, \forall i$, if $\hat{\mathbf{y}}_i = sign(\sum_{i=0}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)) = \mathbf{y}$, then we say that this instance is correct on C_i. Here raises a question.

What if an instance is "correct" on more that one classifiers? We introduce a quantity called confidence, which is $\sum_{i=0}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)$, i.e. the output of the classifier. We choose to believe the classifier with the maximum confidence value to our final result, setting its predicted label as $+1$, and others -1.

1.2 Experiments

Now we enter experiments with algorithms and methods discussed before. Following subsections (1.2.1, 1.2.2, 1.2.3, 1.2.4) describe the results we obtained based on Perceptron with Polynomial Kernel via OvR on hand written digits data set.

1.2.1 Basic Results

This subsection will describe the performance of Perceptron with Polynomial Kernel via table containing averaged training error and testing error on 20 runs on $d = 1, 2, ..., 7$. At the beginning of each run, dataset is randomly divided by training set with number of 80% data points and testing set with the rest. At training stage, as described in Section 1.2.1, we have 10 perceptrons for each d, in total 70 sets of perceptrons. We train them until they "converge" which follows our definition. After the set of perceptron reaches the convergent stage, we re-run our set of perceptron on training dataset to obtain training error. we test the trained model in each run for each d at testing stage. We take the mean and the standard deviation of our training and testing error over 20 runs. The table is presented as follows:

Degree d	**Training error rate(%)**	**Testing error rate(%)**
1	10.204 ± 0.459	8.760 ± 1.052
2	0.710 ± 0.2364	3.322 ± 0.279
3	0.165 ± 0.104	2.781 ± 0.287
4	0.122 ± 0.070	2.751 ± 0.303
5	0.090 ± 0.052	2.754 ± 0.308
6	0.058 ± 0.026	2.816 ± 0.332
7	0.048 ± 0.017	2.907 ± 0.341

Table 1: Training and Testing error rate for 20 runs with different polynomial degree

It can be seen from the table 1.2.1 that, as the polynomial degree d increases, the training error will generally decreases. This can be explained by enlarged hypothesis classes. As polynomial degree grows, the hypothesis classes grows exponentially, thus we will have more H to fit our existing data set, which will lead a decrease in training error. In terms of testing error, the testing error starts to increase at $d = 4$. We infer that this is the results of overfitting to data set when $d > 4$, which can also be explained as the increasing variance. Note that the training error of $d = 1$ is larger than testing error of $d = 1$. We believe this is related with our definition of convergence. When $d = 1$, our definition of convergence is not enough for model to be well trained. In other words, we over-regularised our model, such that it is difficult for our model to perform well on training data set, while the data distribution of testing set is easier for model to classify when $d = 1$.

1.2.2 Cross-validation Results

In this section, we describe the results from K-fold cross validation. Similar to the basic setting, we run the 5-folds validation for 20 times in order to obtain robust results. Dataset is divided by training set with number of 80% data points and testing set with the rest. Different from basic setting, for each polynomial degree $d \in \{1, \cdots, 7\}$, we split training data set to 5 folds. Each time we take 4 folds out of 5 to form a sub training data set and one fold out of 5 to form validation data set. Note that we should do this for 5 times in order to iterate through all data points in training data set. Besides, here we should calculate the Gram matrix for our sub training data set and validation data set, not the whole training set. At the end of each run, we select the optimal d^* that has the smallest validation error. After that, we use it to re-train the whole training data set and test on the testing data set. Since we have 20 independent runs, we will as a result have 20 optimal d^*s and 20 testing error. **The averaged optimal** d **is** 4.85 ± 1.014**, and mean test error rate (%) is** 2.808 ± 0.499. The detailed result is shown as Table 1.2.2.

Run	Optimal d	Test error rate (%)	Run	Optimal d	Test error rate (%)
1	7	2.152	11	5	2.636
2	3	2.259	12	6	2.690
3	5	3.174	13	4	2.690
4	4	2.743	14	5	2.313
5	4	2.582	15	4	2.743
6	6	4.519	16	6	3.281
7	4	2.690	17	6	2.959
8	4	2.582	18	6	2.474
9	4	2.582	19	5	2.797
10	5	2.904	20	4	3.389

Table 2: Optimal d and Testing error rate for 20 runs

1.2.3 Confusion matrix

Confusion matrix (CM) can be derived during the testing stage of Cross validation. First, we construct a confusion matrix filled with all 0s. Every time we have an error in testing stage, we add 1 to corresponding place. Then, we divide the confusion matrix by the number of data points in whole testing set. Thus, the single entry of confusion matrix CM_{ij} represents the ratio between the times of label i is predicted to be label j and the number of data points in whole testing set. The confusion matrix is expressed as:

	0	1	2	3	4	5	6	7	8	9
0	0.0±0.0	0.016±0.025	0.048±0.051	0.019±0.026	0.024±0.04	0.024±0.027	0.024±0.032	0.0±0.0	0.013±0.023	0.008±0.019
1	0.0±0.0	0.0±0.0	0.003±0.012	0.003±0.012	0.032±0.046	0.003±0.012	0.035±0.031	0.011±0.027	0.013±0.029	0.013±0.023
2	0.035±0.035	0.024±0.032	0.0±0.0	0.056±0.05	0.067±0.078	0.008±0.026	0.027±0.05	0.046±0.052	0.032±0.036	0.003±0.012
3	0.022±0.036	0.016±0.034	0.046±0.039	0.0±0.0	0.008±0.019	0.134±0.065	0.011±0.027	0.043±0.05	0.083±0.067	0.011±0.022
4	0.008±0.019	0.051±0.055	0.03±0.036	0.005±0.016	0.0±0.0	0.011±0.022	0.035±0.043	0.013±0.029	0.008±0.026	0.105±0.055
5	0.059±0.061	0.013±0.029	0.035±0.035	0.113±0.076	0.043±0.055	0.0±0.0	0.054±0.061	0.005±0.023	0.035±0.049	0.054±0.045
6	0.067±0.045	0.019±0.026	0.016±0.025	0.0±0.0	0.043±0.032	0.03±0.06	0.0±0.0	0.003±0.012	0.024±0.032	0.005±0.016
7	0.0±0.0	0.019±0.031	0.043±0.036	0.011±0.022	0.054±0.051	0.008±0.019	0.0±0.0	0.0±0.0	0.019±0.039	0.11±0.081
8	0.067±0.033	0.059±0.045	0.043±0.055	0.11±0.073	0.027±0.036	0.083±0.073	0.016±0.025	0.022±0.031	0.0±0.0	0.019±0.035
9	0.019±0.031	0.011±0.022	0.003±0.012	0.022±0.043	0.075±0.055	0.005±0.016	0.005±0.016	0.081±0.077	0.013±0.033	0.0±0.0

Table 3: Confusion Matrix (%)

While the heat map of confusion matrix is shown as below in figure 1:

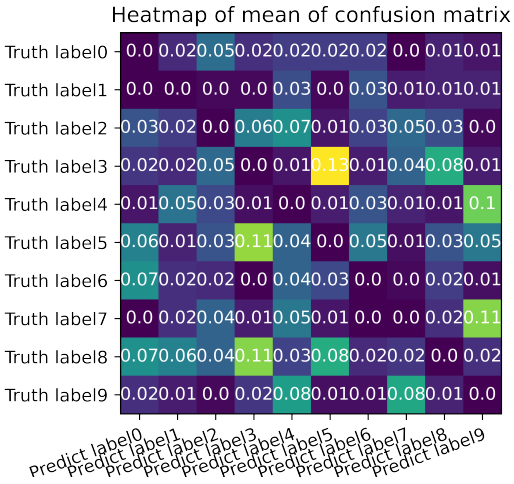

Figure 1: Heatmap of confusion matrix, averaged test error between labels

1.2.4 Five hardest label

During the training and testing processes, there are five hardest labels to predict. We define the hardest as following: 1. The specific instance is predicted wrong in most of the case. 2. The specific instance is predicted with large confidence value. The corresponding intuition behind this is that: 1. Perceptron cannot identify the specific instance in most of cases. 2.Perceptron strongly believe that such instance is belong to other labels. We select the instance with the most error cases as well as the highest confidence value. The five hardest label is shown as Figure 2:

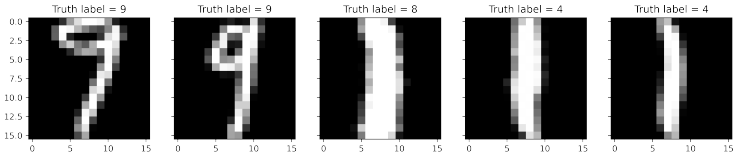

Figure 2: The five hardest label with misclassified times of 4, 4, 6, 10, 20(from left to right)

It can be seen that it is common hard for perceptron to classify these labels. For the first instance with label 9, the width of this "9" is large on the head and it is not that straight. It is hard for perceptron to classify the first instance from 9 to 7. For three instances at the right, it is extremely not clear for perceptron to identify them from their truth label to label 1.

1.3 Perceptron with Gaussian Kernel

In this section, we will describe Perceptron with Gaussian Kernel via OvR on Hand written digit data set. Furthermore, we will compare the performance of Gaussian Kernel vs Polynomial Kernel via time consuming on training one epoch as well as best testing error on whole data set. The Gaussian Kernel is expressed as following:

$$\mathbf{K(p,\ q)} = e^{-c||\mathbf{p}\ -\mathbf{q}||^2}$$

We use guassian kernel to replace polynomial kernel in basic setting. The hyperparameter in Guassian Kernel changes from d to c. The first thing we do is to test the impact of setting different c on performance of the perceptron. Similar to Polynoimal kernel with basic results, we run 20 times on different c in order to obtain robust results, which can be seen as following table:

c	**Training error rate**(%)	**Testing error rate**(%)
10^{-5}	34.168 ± 1.166	16.611 ± 4.195
10^{-4}	23.570 ± 2.655	12.087 ± 2.133
10^{-3}	10.280 ± 1.643	6.181 ± 0.961
10^{-2}	1.794 ± 0.368	2.593 ± 0.387
10^{-1}	2.056 ± 0.420	5.153 ± 0.605
1	2.423 ± 0.296	6.686 ± 0.558
10^{1}	8.224 ± 1.213	19.747 ± 0.747
10^{2}	32.918 ± 5.973	71.393 ± 0.726
10^{3}	49.977 ± 0.006	82.856 ± 0.648
10^{4}	50.000 ± 0.000	82.969 ± 0.638

Table 4: Training and Testing error rate for 20 runs with different guassian c

It can be seen from table 3 that, when c is near 10^{-2}, the performance of perceptron reaches its peak. Thus, we choose the region of c to be $0.010, 0.011, ...0.030$ to cross validate over.

Gaussian Kernel with K-cross validation Similar to the setting that we apply to Polynomial Kernel, we still use 5-fold cross validation to choose the optimal c^* for 20 independent times. Thus we have 20 c^* and 20 best test error, which is shown in table:

Run	**Optimal c**	**Test error rate**	**Run**	**Optimal c**	**Test error rate**
1	0.011	2.259	11	0.015	2.097
2	0.016	2.636	12	0.013	2.797
3	0.012	2.259	13	0.018	3.012
4	0.010	2.367	14	0.02	2.205
5	0.019	3.227	15	0.015	2.851
6	0.022	2.313	16	0.023	2.474
7	0.016	2.152	17	0.017	2.797
8	0.019	2.098	18	0.022	3.228
9	0.017	2.743	19	0.013	3.227
10	0.013	3.335	20	0.018	2.528

Table 5: Optimal c and Testing error rate for 20 runs of Perceptron with Gaussian Kernel

The averaged optimal c is 0.01645 ± 0.00365 and the mean test error (%) is 2.630 ± 0.405. It can be seen from the Table 1.3 that the variance is large on Gaussian Kernel. In other words, gaussian kernel has unstable hypothesis class for this specific data set, which is also explainable given that gaussian kernel has infinite VC-dimension. It means that we are finding the optimal hyperplane in a feature space that has infinite dimension.

Comparison to Polynomial Kernel We compare the Gaussian Kernel and Polynomial Kernel via their learning efficiency as well as their averaged best test error. Based on our previous definition of "convergence" of perceptron, we run 10 independent times to see the level of epochs for two kinds of perceptron to converge. Results shows that perceptron with polynomial kernel needs 8.4 epochs on average while the one with Guassian kernel only requires 7.4 epochs on average. The results show that perceptron with Guassian kernel is able to learn faster than the one with polynomial kernel. It is intuitive to regard this result as the more learnable effect of dimensional space created by Guassian kernel. In terms of their best test error, Gaussian Kernel has an averaged minimum test error of 2.630%, compared to polynomial kernel with test error of 2.808%. Both results show that Guassian kernel has a better performance on this specific hand written digit data set.

1.4 One versus One (OvO) Method

We will describe our alternative method which is also able to generalise our binary perceptron to K-classifier.

Intuition of One versus One With K classes to classify, we may construct $K * (K-1)/2$ classifiers where each classifier is responsible to compare one single pair of labels selected from our K-classes without repeat cases. For example, in our case, we have 10 classes, 10*9 = 45 classifiers will be trained. For each classifier, it will have the first element as its "main target label", e.g., classifier (1,2) is responsible for classifying label 1 as +1 and label 2 as -1 from a mixture of label 1 and label 2. When a new instance x enter the model, 45 classifiers regardless of their "main target label" will vote for this instance. Thus, we use the result of voting to make our decision. One disadvantage to apply OvO compared to OvR is obvious. It requires $\frac{k^2-k}{2}$ classifiers compared to k classifier. It brings about side effects such as slow-speed computation, large memory requirement when applying to complex tasks.

Implementation details of OvO In our implementation, we make **two improvements** on this method.

- It is slow to compute all the classifier in serial. In fact, classifiers can be represented by vector α since $\hat{y}_t = sign(\sum_{i=0}^{n} \alpha_i K(x_i, x_t))$, where Kernel matrix is fixed for training/testing stage. Thus, we construct "alpha matrix" $\alpha_{MAT} \in \mathbb{R}^{(\frac{k^2-k}{2}) * m_{train}}$. In our case, $\alpha_{MAT} \in \mathbb{R}^{45 * m_{train}}$. Thus, prediction can be made by computing in parallel, which is shown as:

  $$\hat{y} = sign(\alpha_{MAT} * K_i^T)$$

  where K_i can be i-th row of K_{train} or K_{test}, $\hat{y} \in \mathbb{R}^{\frac{k^2-k}{2}}$, $*$ represents the inner product, m represents number of training/testing data points.

- Previously we mentioned that all classifier will vote, thus $\frac{k^2-k}{2}$ votes will be made. The final number of classes in our task is k. To change $\frac{k^2-k}{2}$ to k, we are inspired that matrix multiplication can change the dimension of matrix. Thus, we define our manipulation matrix to be $M \in \mathbb{R}^{\frac{k^2-k}{2}}$. Whenever we have $\hat{y} \in \mathbb{R}^{\frac{k^2-k}{2}}$, we can derive the voting results by

$$y_{pred} = argmax(M * \hat{y})$$

  where $*$ represents the inner product.

Whole process of OvO can be found in algorithm 3

Algorithm 3 One Versus One

Input: alpha matrix α_{MAT}, manipulation matrix M
Output: test mistakes
Data: $K_{train}, K_{test}, y_{train}, y_{test}$
/* start training */
for *i in length*(K_{train}) **do**
  $\hat{y} = sign(\alpha_{MAT} * K_{train}[i]^T)$ // *obtain the voting of $\frac{k^2-k}{2}$ classifier, $\hat{y} \in R^{\frac{k^2-k}{2}}$
  $y_{pred} = argmax(M * \hat{y})$ // * count voting results using manipulation matrix, $y_{pred} \in R^k$
  if $argmax(y_{pred}) \neq y_{train}[i]$ **then**
    update corresponding classifier
/* start testing */
test mistake = 0
for *i in length*(K_{train}) **do**
  $\hat{y} = sign(\alpha_{MAT} * K_{test}[i]^T)$
  $y_{pred} = argmax(M * \hat{y})$
  if $argmax(y_{pred}) \neq y_{test}[i]$ **then**
    test mistake + 1

1.4.1 Basic Results

Similar to the setting of Polynomial Kernel based on OvR method, we use OvO instead.
We run 20 times for observing the result with d in range 1-7. Table 1.4.1shows the results as following:

Polynomial degree d	Training error rate(%)	Testing error rate(%)
1	2.667 ± 0.736	6.277 ± 0.782
2	0.116 ± 0.078	3.386 ± 0.401
3	0.061 ± 0.038	3.136 ± 0.329
4	0.051 ± 0.027	3.066 ± 0.326
5	0.037 ± 0.023	3.166 ± 0.343
6	0.034 ± 0.025	3.337 ± 0.395
7	0.032 ± 0.024	3.513 ± 0.445

Table 6: Training and Testing error rate for 20 runs with different polynomial degree (OvO)

It can be observed that, there is a decline trend on the performance of OvO compared to OvR (table1.2.1 and table1.4.1) over all polynomial degree (expect for $d = 1$, when $d = 1$, we have more "experts" than that in OvR, which will result a better performance even if each of expert is not well trained, as the theory of bagging indicated[4].). Our intuition is that, there are lots of "bad" instances inside the data set. In other words, there are many similar instances with different labels (E.g. label 4 and label 1 shown in section 1.3.4) inside data set, thus when a specific instance with truth label (say label 4) is voted among $k * (k-1)/2$ classifiers, there are a number of irrelevant classifiers vote for their main labels (classifiers with main target label 1). Then, for this specific instance, there is a possible case that this instance is voted more for label 1 instead of label 4, which will result as a mistake. Yet this situation will rarely happen on OvR, since in OvR, it compares one label to the rest. Thus it uses the global information of data set, **which is more helpful in this specific hand written digit data set.** While in most cases, OvO will outperform OvR since it only uses corresponding information to train corresponding classifiers, which will prohibit the influence of noisy information.

1.4.2 Cross-Validation Results

Similar to the setting of polynomial perceptron using OvR, we cross validation to find our optimal d for 20 independent runs. Results are shown in Table 1.4.2:

Run	Optimal d	Test error rate	Run	Optimal d	Test error rate
1	5	0.031	11	5	0.029
2	5	0.024	12	5	0.034
3	3	0.040	13	3	0.028
4	3	0.037	14	3	0.036
5	4	0.035	15	4	0.032
6	3	0.028	16	4	0.034
7	4	0.032	17	3	0.030
8	5	0.025	18	5	0.032
9	4	0.031	19	3	0.028
10	3	0.036	20	4	0.030

Table 7: Optimal d and Test error rate for 20 runs of Perceptron with Polynomial Kernel (OvO)

The averaged optimal d is 3.9 ± 0.831 and the mean test error (%) is 3.171 ± 0.395. It can be seen in Table 1.4.2 and averaged results that, compared to OvR, the performance of OvO is slightly lower, while the optimal averaged d^* in OvO is smaller than that in OvR ($d* = 4.85$). Our intuition behind this is that, from the perspective of bagging, our classifier can be simpler (less hypothesis classes can be enough) given that we have more classifier and each classifier is responsible for classifying fewer instances.

2 Part 2

2.1 Implementation

In this part we design several functions to help us implement the spectral clustering algorithm. Following the instruction, the complete algorithm is shown below:

Algorithm 4 Spectral Clustering

Result: $\hat{y}$
$y \leftarrow \{-1, 1\}^m$; $X \leftarrow \{\mathbb{R} \times \mathbb{R}\}^m$; $c \leftarrow \mathbb{R}$;
$W \leftarrow [\exp(-c\|x_i - x_j\|^2)]_{i,j=1,1}^{m,m}$;
$i \leftarrow 0$; $j \leftarrow 0$;
while $i < m$ **do**
  while $j < m$ **do**
    if $i == j$ **then**
      $D_{ij} \leftarrow \sum_{k=1}^{m} W_{ik}$
    else
      $D_{ij} \leftarrow 0$
    $j \leftarrow j + 1$
  $i \leftarrow i + 1$
$L \leftarrow D - W$
$v_2 \leftarrow$ eigenvector corresponding to the second smallest eigenvalue of L;
$i \leftarrow 0$;
while i ¡ m **do**
  if $v_2[i] < 0$ **then**
    $\hat{y}[i] \leftarrow 1$
  else
    $\hat{y}[i] \leftarrow -1$
  $i \leftarrow i + 1$

The first self-designed function is `W_matrix`, which takes x data and c value as its inputs. It will return the weight matrix W with given weight formula $W_{ij} = \exp(-c\|x_i - x_j\|^2)$. Next function is `L_matrix`, which takes weight matrix as its input. This function will return graph Laplacian $L = D - W$. With calculated L, we could finally finish the clustering algorithm by the function `cluster`. It accepts L and return the predicted y value $\hat{y}$. Inside the function, it first gets eigenvalues and corresponding eigenvectors. Then we locate the eigenvector v_2 whose corresponding eigenvalue is the second small one. We predict $\hat{y}[i]$ as $sign(v_2)[i]$. Apart from these three functions about Spectral Clustering, we have another function `plot_best_figure` for the first two experiments. With given range of $c \in \{2^i : i \in \{-10.0, -9.9, \cdots, 0, 0.1, \cdots, 9.9, 10.0\}\}$, this function will return parameter c which could correctly or almost correctly cluster data and its corresponding $\hat{y}$. Besides, this function will plot the clustered data. From Figure 3 and 4, we could see that when $c = 2^{4.5} \simeq 22.627$, this dataset can be perfectly clustered. And Figure 5 and 6 show results about the second experiments. This time when $c = 2^{3.6} \simeq 12.126$, our self generated dataset can be correctly clustered.

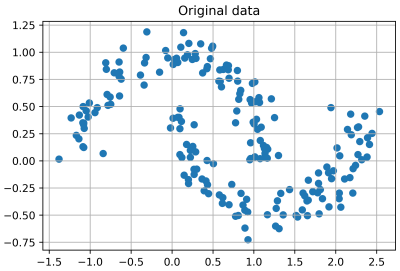

Figure 3: Original Plot of "twomoon"

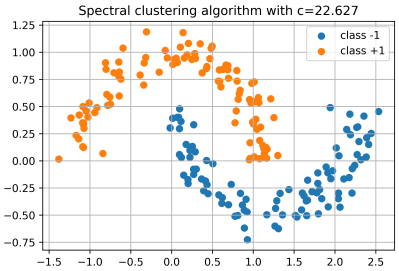

Figure 4: Clustered "twomoon" plot

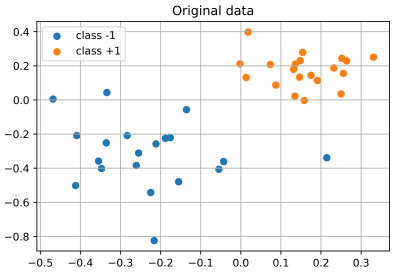

Figure 5: Original plot of self-generated data

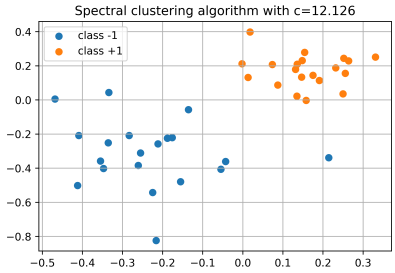

Figure 6: Clustered self-generated plot

In the experiment 3, we change the range of c as: $c \in \{0.000, 0.001, \cdots, 0.100\}$. After we get the predicted y values $\hat{y}$, we implement CP measurement with the following algorithm:

Algorithm 5 CP Evaluation

Result: *cp_list*

$y \leftarrow \{-1, 1\}^m$; $c \leftarrow \mathbb{R}$;

$i \leftarrow 0$;

while i ¡ m **do**

  $classy[i] \leftarrow 2 * (y[i] == 1) - 1$;

  $i \leftarrow i + 1$

$i \leftarrow 0$;$l\_pos \leftarrow 0$;$l\_neg \leftarrow 0$;

while i ¡ m **do**

  if $\hat{y} == classy[i]$ **then**

    $l\_pos \leftarrow l\_pos + 1$

  else

    $l\_neg \leftarrow l\_neg + 1$

  $i \leftarrow i + 1$

$cp\_list.append(\frac{\max(l\_pos, l\_neg)}{m})$

Generally speaking, CP measures the maximum number of either correctly or wrongly clustered points. The final result is the ratio with the maximum number over the total sample number. We plot $CP(c)$ as a function of c to locate a possible optimal c.

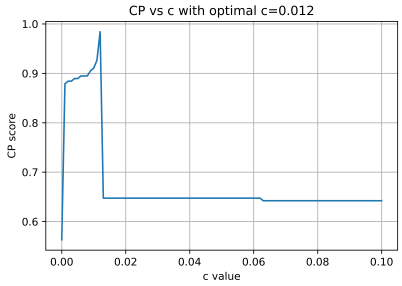

Figure 7: CP score vs c

From the figure we could roughly see that all CP score is larger than 0.55. Actually $CP(c)$ is bounded in $[0.5, 1]$. For convenience, in the latter explanation, we use ℓ_-, ℓ_+, ℓ representing $|\ell_-|, |\ell_+|, |\ell|$. The upper bound is easy to explain: as $\ell_+ \leq \ell$ and $\ell_- \leq \ell$, then $\max\{\ell_-, \ell_+\} \leq \ell \Rightarrow CP = \frac{\max\{\ell_-, \ell_+\}}{\ell} \leq 1$. For the lower bound:

- if $\ell_- < \frac{1}{2}\ell$, then $\ell_+ = \ell - \ell_- > \frac{1}{2}\ell$, so $\max\{\ell_-, \ell_+\} = \ell_+ > \frac{1}{2} \Rightarrow CP = \frac{\max\{\ell_-, \ell_+\}}{\ell} > \frac{1}{2}$;
- if $\ell_- > \frac{1}{2}\ell$, then $\ell_+ = \ell - \ell_- < \frac{1}{2}\ell$, so $\max\{\ell_-, \ell_+\} = \ell_- > \frac{1}{2} \Rightarrow CP = \frac{\max\{\ell_-, \ell_+\}}{\ell} > \frac{1}{2}$;
- if $\ell_- = \frac{1}{2}\ell$, then $\ell_+ = \ell - \ell_- = \frac{1}{2}\ell$, so $\max\{\ell_-, \ell_+\} = \ell_+ = \frac{1}{2} \Rightarrow CP = \frac{\max\{\ell_-, \ell_+\}}{\ell} = \frac{1}{2}$.

Therefore, we conclude CP is bounded below by $\frac{1}{2}$.

2.2 Questions

2.2.1 Question 1

The basic idea of $CP(c)$ is to measure the percentage of the major group. There are two situations we are concern about:

1) If the major group is ℓ_+ (i.e. $\ell_+ >= \frac{1}{2}\ell$). Then $\max\{\ell_-, \ell_+\} = \ell_+$. So more samples in ℓ_+ means more correctly clustered data, which shows a better parameter c.

2) If the major group is ℓ_- (i.e. $\ell_- >= \frac{1}{2}\ell$). Then $\max\{\ell_-, \ell_+\} = \ell_-$. However, since the task is binary clustering, we just need to switch the labels of two classes can "make" ℓ_- to be ℓ_+. Therefore, under this situation, a bigger $CP(c)$ score still means a better choice of c

To sum up, in the binary clustering task, $CP(c)$ is concerning about whether a c can result to a major group, hence it is a reasonable measurement. Furthermore, we are looking for $\arg\max_c CP(c)$.

2.2.2 Question 2

Claim: The first eigenvalue of the Laplacian is zero and the corresponding eigenvector is the constant vector.

Proof. By definition, we could write L as:

$$L = \begin{pmatrix} \sum_{i=1}^m w_{1i} - w_{11} & -w_{12} & \dots & -w_{1m} \\ -w_{21} & \sum_{i=1}^m w_{2i} - w_{22} & \dots & -w_{2m} \\ \dots & \dots & \dots & \dots \\ -w_{m1} & -w_{m2} & \dots & \sum_{i=1}^m w_{mi} - w_{mm} \end{pmatrix}$$

where $w_{ij} = \exp(-c\|x_i - x_j\|^2), \forall i, j \in \{1, \cdots, m\}$.
Hence, for any $j \in \{1, \cdots, m\}$, we observe that $(\sum_{i=1}^{m} w_{ji} - w_{jj}) - \sum_{i=2}^{m} w_{ji} = 0$, i.e. the sum of each row is 0 respectively.
Therefore, there exists a $m * 1$ constant vector $\mathbb{1} = (1, \cdots, 1)^T$ s.t. $L\mathbb{1} = \mathbf{0}$.
By definition, the eigenvalue λ is the number s.t. $L\mathbf{x} = \lambda\mathbf{x} \Rightarrow (L - \lambda I)\mathbf{x} = \mathbf{0}$.
Let $\mathbb{1}$ be the eigenvector $\mathbf{x}$, then we get $L - \lambda I = L \Rightarrow \lambda = 0$. □

2.2.3 Question 3

The reason why Spectral Clustering algorithm works could be explained by graph perspective. The notation of W and D matrices are the same as the definition in the graph theory. If we consider a fully connected graph $G = (V, E)$ where V represents vertices, the set of our input x's, and E represents edges between each pair of vertices with weight $w_{ij}, \forall i, j \in \{1, \cdots, m\}$. All these weights construct the weight matrix W, and D matrix is the degree matrix, where there are m degree values corresponding to m vertices. Therefore, the question could be restated as finding a mincut which partitions graph G into 2 different groups [6]. The refined RatioCut objective function could be imported to generate two large groups based on similarity [1]. While this problem can be fully transferred to relaxed optimization problem as [1] indicated, which is shown as:

$$\min_{f \in \mathbb{R}^n} f^{'} L f \text{subject to } f \perp \mathbb{1}, \|f\| = \sqrt{n}$$

Together with Rayleigh-Ritz theorem, we could approximate the minimizer of L with the eigenvector corresponding to the second smallest eigenvalue. [5]

2.2.4 Question 4

Before c reaches the optimal:
Since $w_{ij} = e^{-c\|x_i - x_j\|^2}$, we can obeserve that w_{ij} is descreaing as c grows. We know that L is

$$\begin{pmatrix} \sum_{i=1}^{m} w_{1i} - w_{11} & -w_{12} & \dots & -w_{1m} \\ -w_{21} & \sum_{i=1}^{m} w_{2i} - w_{22} & \dots & -w_{2m} \\ \dots & \dots & \dots & \dots \\ -w_{m1} & -w_{m2} & \dots & \sum_{i=1}^{m} w_{mi} - w_{mm} \end{pmatrix}$$
$$= \begin{pmatrix} \sum_{i \in \{1, \cdots, m\} \setminus \{1\}} w_{1i} & -w_{12} & \dots & -w_{1m} \\ -w_{21} & \sum_{i \in \{1, \cdots, m\} \setminus \{2\}} w_{2i} & \dots & -w_{2m} \\ \dots & \dots & \dots & \dots \\ -w_{m1} & -w_{m2} & \dots & \sum_{i \in \{1, \cdots, m\} \setminus \{m\}} w_{mi} \end{pmatrix}$$

So when $w > 0$ is decreasing, $\|L\|_F$ is decreasing.
Since W is symmetric, D as a diagonal matrix is also symmetric.
Then $L = D - W$ is also symmetric.
As $L = Q\Lambda Q$, $\|\Lambda\|_F \propto \|L\|_F$ with property of symmetric matrix, in other words, the F-norm of any matrix will not be influence by orthogonal transformation
$\Rightarrow \|\Lambda\|_F$ is decreasing as c is increasing .
Observe from figure 8 that as $\|\Lambda\|_F$ decreases, every eigenvalue is decreasing.
So we conclude that the second smallest eigenvalue is also decreasing.
From Luxburg [6] it is equivalent to cut is decreasing, hence we show that the clustering is better.

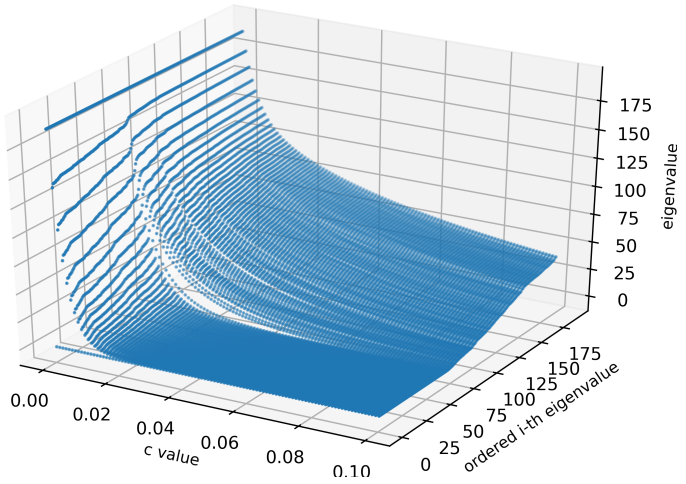

Figure 8: Change of eigenvalues w.r.t. c value

3 Part 3

In this part, we will be presenting our measurement of sample complexity of four learning algorithms, Perceptron, Winnow, Least Square and 1 Nearest Neighbor .We will divide our report to two parts for better presentation.

i. Implementation

ii. Questions

3.1 Implementation

3.1.1 Part a. 'Fake' simulation results

Our results of sample complexity (SC) for four algorithms can be seen as in figure 9, 10, 11 and 12. Note that except 1-NN with dimension measurement stopped at $n = 15$ due to the high time complexity, other three algorithms are measured to $n = 100$.

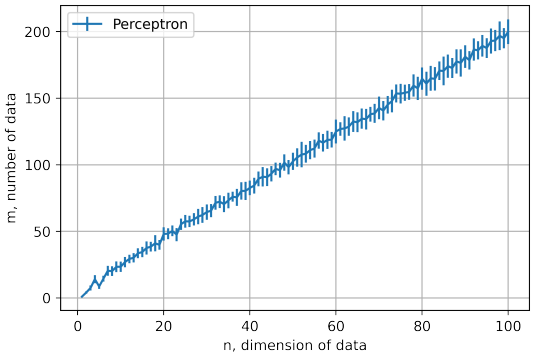

Figure 9: SC plot of Perceptron

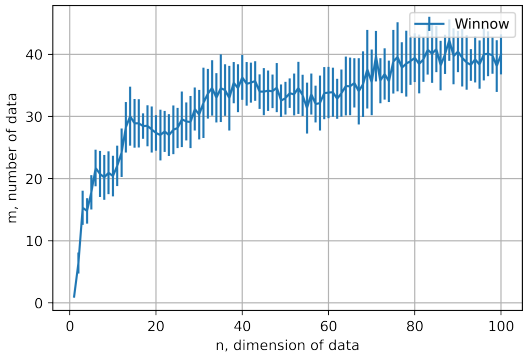

Figure 10: SC plot of Winnow

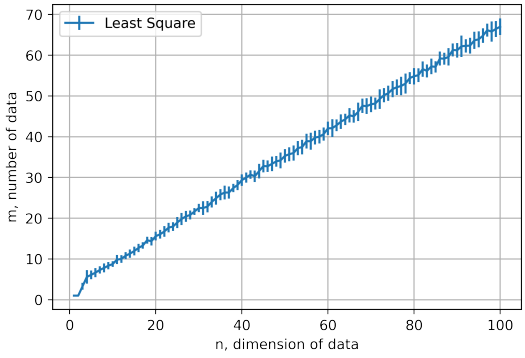

Figure 11: SC plot of Least Square

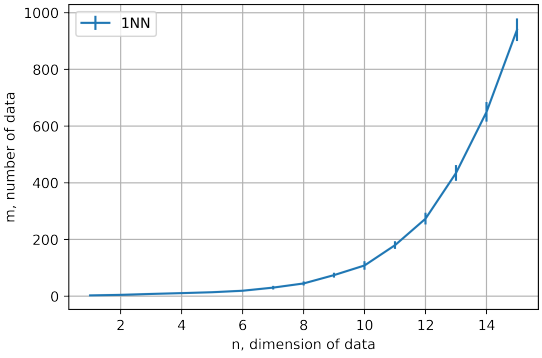

Figure 12: SC plot of 1 Nearest Neighbor

3.1.2 Part b. Implementation Details and Trade Off

Implementation details Due to the limitation of space, we do not describe the detailed implementation of four algorithms. Instead, we focus on the implementation details of how we derive the sample complexity of these algorithms. The whole process (we call it a pseudo algorithm) of obtaining sample complexity for any algorithm is shown as Pseudo Algorithm 6.

Algorithm 6 Sample Complexity computation

Input: Algorithm A, dimension range n_{range}
Output: Averaged sample complexity list m_{list}

```
/* n_range refers to the dimension range we want to test, starting from 1 */
/* start training */
1  avm_list = []
   for runs in range(20) do
2      for n in range(n_range) do
3          m_list = []
               /* every time we test a new dimension n */
           /* we start from testing one sample and adding up */
4          m = 1
           while True do
               /* generate training/testing data set corresponding to n and m */
               /* X_train has m samples, each sample has n dimensions */
               /* X_test has 10000 samples, aiming to be relatively large to represent distribution
                  of data */
5              X_train, X_test, y_train, y_test = data(m, n)
                  /* use generated data to obtain generalised error of algorithm A */
6              error rate = A(X_train, X_test, y_train, y_test)
               if error rate ≤ 10% then
7                  times = 0
                   for i in range(5) do
8                      repeat code line 3, 4 // *use current m to obtain error rate for 5 times
9                      if error rate < 10% then
10                         times + 1
11                 if times == 5 then
                       /* All 5 runs agree that current m will have generalised error less than 10%
                          */
                       /* Then we store this m to corresponding n */
12                     m_list.append(m)
13             else
14                 m + 1 // * add samples to see if it can satisfy 10% generalised error for 5
                      times
15     avm_list.append(m_list)
16 return mean(avm_list)
```

Note that it is not possible to iterate through all possible combination of data (2^npossible combinations) as definition of generalisation error indicated, we approximate the results by randomly generating our data and take the averaged results. We choose our test size equal to 10000, aiming to be relatively representative to the data distribution. As described in Algorithm 6, for every dimension we want to test for sample complexity, we add our number of sample m from $m = 1$. Once we find out that current m can be enough for our algorithm to have less that 10% generalised error, we continue to use current m to generate data freshly and test for other five times. If our algorithm have generalised error less than 10% on all these five refreshed data set, then we regard the current sample size m as a relative stable sample size for this specific data dimension n. Thus, we can move on to test for next data dimension $n+1$. We run this whole process for 20 times and take the mean value of sample complexity.

Trade Off We elaborate the trade off of our method from the perspective of pros and cons of our method.
PROs: First, we run for 20 times and take the mean of our sample complexity. This enables us to explore more possible permutations of our randomly generated data. Thus, more exploration on possible permutation, more stable results we will obtain. Second, every time we test a new dimension by setting $m = 1$, which means that we don't have the assumption that increasing n will cause increasing m, since data is randomly generated. This will enable us to be more accurate in terms of randomness. In other words, this enables us to explore far more possible permutation of data. Third, every time we found our possible sample complexity, we repeat 5 times to obtain further stable results on specific sample complexity for our specific data dimension. Same as before, more permutations can be explored at this stage. In conclusion, we try to be as accurate as we can, with the limitation of computational and time resource. The relatively small variance bar on each plot proves our intuition of PROs of our method.
CONs: Disadvantages are relatively obvious. Given that we have tried our best to simulate as many times as we can (limit on computational resource and time) to cover all possible data permutations, we have to spend much time on simulating the mean results.

3.2 Question

3.2.1 Part c. Estimated relationship between m and n for 4 algorithms

We will divide our analysis of the relationship by algorithms.

Perceptron It can be seen from [2] that, a binary learner with PAC learnable hypothesis class (in realisable case) has an upper bound and lower bound of sample complexity as shown below:

$$C_1 \frac{VCdim(H) + \log(1/\sigma)}{\epsilon} \leq m_H(\epsilon, \sigma) \leq C_2 \frac{VCdim(H)\log(1/\epsilon) + \log(1/\sigma)}{\epsilon}$$

where $1-\sigma$ represents for our confidence of this derived upper bound and lower bound, ϵ represents for the generalised error rate of our algorithm, $|H|$ represents the size of our hypothesis class.

Since our data set is generated as $x \in \{1, -1\}^n, y \in \{1, -1\}$, **it is obvious to see that our data set is separable, which satisfies the realisable case setting.** In perceptron, when we increase the dimension of the data, we increase the size of our weight w, since $y = \sum_i w_i * x_i$, where $w_i \in \{-1, +1\}^2$. Thus it is obvious that $|H|$ in perceptron grows as 2^n. Thus, it is obvious that

the VC dimension of perceptron is n. By fixing $\epsilon = 10\%$, we can derive our sample complexity for perceptron are bounded as

$$\lambda_1 n \leq m_{preceptron}(0.1, \sigma) \leq \lambda_2 n$$

where λ_1 and λ_2 represents to constanct. In this way, it is obvious to see that when n linearly grows, m will linearly grow as a result of derived upper and lower bound of sample complexity. Thus, $m(n) = \Theta(n)$

Winnow From [3] we can see that, the mistake upper bound of winnow is derived as:

$$M \leq 3k(\log n + 1) + 2$$

where k represents for k literal conjunctions. It can be any number less than n in our case. Thus, if we divide sample complexity at both sides, it will shown as:

$$\frac{M}{m} \leq \frac{3k(\log n + 1) + 2}{m}$$

If we fix $\frac{M}{m}$ to be generalised error in our case, then m will grow linearly as n grows exponentially. Note that in this case $m(n)$ has a lower bound, since we have to meet M mistakes in advance, we can gaurrantee that generalised error is less that 10%, thus $m(n) = \Omega(\log(n))$

Least Square From [2] we can see that, the VC dimension of Halfspaces:

$$H = \{\mathbf{x} \Rightarrow sign(\mathbf{W^*X}) : W \in \mathbb{R}^n\}$$

is 2^n. Thus we can easily derive our relationship of m and n as $m(n) = \Theta(n)$.

1 Nearest Neighbor From the perspective of learning theory, 1-NN is not really learning something from our training data but just remember the training data set. In our case, where $x \in \{-1, +1\}^n$, whenever a new instance comes in for testing, 1-NN will just search for the nearest point of our new instance. In other words, if we decrease the generalised error to be 0, then the sample complexity will be bounded by 2^n, which is the whole permutation of our possible data points in training set, thus to ensure any incoming testing instance can find the nearest neighbor and predict will 0 error. In this way, we can conclude that the relationship between m and n is $m = \Omega(2^n)$.

3.2.2 Part d.

From [3], we can see that, **For any separable data set**, we can derive an upper bound of mistakes (M) that perceptron can make by viewing as:

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

where R represents for the radius of hyperball that can cover all data points, γ represents for the smallest distance between the data point in data set and separable hyperplane. Since our data set is generated as $x \in \{1, -1\}^n, y \in \{1, -1\}$, **it is obvious to see that our data set is separable, which is satisfied for our mistake bound assumption**, since every label (y_i) is obtained by selecting the entry of first dimension of our instance (x_i). In this case, not only assumption is satisfied, R^2 and γ^2 can also be easily derived as n, 1 respectively. For $R^2 = n$, it is obvious to see

since $x \in \{1, -1\}^n$. For $\gamma^2 = 1$, this is because in any dimension n, we can choose the hyperplane formed by other coordinate axes expect for the coordinate corresponding to the first dimension to completely separate our data set. In this case, the smallest distance is the absolute value of the first dimension of the data, which is 1 in all cases. Thus, we can derive our mistake upper bound by $M \leq n$.
Now we have derived that our mistake upper bound in this case is n, which is the size of dimension of our data. We uniformly sample an integer $s \in \{1, 2, ..., m\}$ to train our perceptron on $(x_1, y_1), (x_2, y_2), ..., (x_s, y_s)$. We want to find the upper bound $\hat{p}_{m,n}$ on the probability that the perceptron will make a mistake on s-th sample given that we have our mistake upper bound n. So we allocate these n mistakes inside our data set with size m. These n mistakes are distributed close with each other, i.e. $x_i, x_{i+1}, ..., x_{i+n-1}$ is the mistake point for the sake of "upper bound", we call this "mistake" region. If we select s inside mistake region, then (s+1)-th point will be a mistake. Thus, our question transfers to: what is the probability of our s lying in mistake region? Given that s is uniformly selected from $\{1, 2, ..., m\}$, then our $\hat{p}_{m,n} = \frac{n}{m}$. If n is larger than m, then our upper bound will be 1 in this case.
In conclusion, our upper bound of $\hat{p}_{m,n}$ is $\hat{p}_{m,n} = min(1, \frac{n}{m})$

References

[1] Lars Hagen and Andrew B Kahng. "New spectral methods for ratio cut partitioning and clustering". In: *IEEE transactions on computer-aided design of integrated circuits and systems* 11.9 (1992), pp. 1074–1085.

[2] Mark Herbster. "Learning Theory COMP0078: Supervised Learning". In: (2021).

[3] Mark Herbster. "Online Learning I COMP0078: Supervised Learning". In: (2021).

[4] Mark Herbster. "Tree based learning and ensemble methods COMP0078: Supervised Learning". In: (2021).

[5] H Lütkepohl. "Handbook of matrices. 1996". In: *John wiley&sons* ().

[6] Ulrike von Luxburg. "A Tutorial on Spectral Clustering". In: *CoRR* abs/0711.0189 (2007). arXiv: 0711.0189. URL: http://arxiv.org/abs/0711.0189.