Supervised Learning (COMP0078) – Coursework 2

Candidate Number: 18145066, 19019509

December 13, 2022

We implemented the answers for each parts in three different files, "Part_one.py", "Part_two.py" and "Part_three.py". To see the result for each question, just simply execute the "CW2.ipynb" Jupyter notebook. All of these three parts are already imported in the notebook.

1 Part 1 Kernel perceptron

1) One versus rest (OvR) is the first method for multi-class classification using a two-class classifier. The purpose of this approach is to divide the multi-class dataset into multiple binary classification datasets, each distinguishing one class from the rest, i.e. "Class r" and "not Class r". A binary classifier is then trained on each binary classification problem, and the most confident model is used to make predictions. The OvR algorithm is described in details below:

Algorithm 1 OvR Kernel Perceptron

Input: $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\} \in (\mathfrak{R}^n, \{0 \ldots R\})^m$

Compute the kernel matrix $K_{ij} = K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$ for $\forall x_i, x_j \in \mathbf{X}$
$\mathbf{W} \leftarrow \{0\}^{R \cdot m}$
for epoch $\leq$ max_epoches **do**
  for $t \leq m$ **do**
    $\hat{y} \leftarrow \arg\max W \cdot K(\cdot, \boldsymbol{x}_t)$
    if $\hat{y} \neq y_t$ **then**
      $W_{y,t} \leftarrow 1$
      $W_{\hat{y},t} \leftarrow -1$
    end if
  end for
end for

Instead of determining the label with the sign of the outcome, each two-class classifier in our model gives a value that is equal to the outcome $\mathbf{w}_t(\mathbf{x}_t)$ as a sort of confidence value. The classifier with the highest confidence wins the vote and the label is then predicted to be the true class of that classifier. To avoid unnecessary loops, we pre-computed the kernel $K(xi, xj), \forall i, j \in m$, where m is the size of the "batched" training set. We then walk through each row in the kernel matrix in a random order, giving us $K(xi, \cdot)$ for each i. Since the new weight is the sum of $\boldsymbol{\alpha}_t K(\mathbf{x}_t, \cdot)$ for all previous t's, the computation can be simplified by matrix multiplication of a matrix of $\boldsymbol{\alpha}_t$ and a matrix of $K(xi, \cdot)$. Thus we

create a weight matrix of zeros with dimension $R \times m$, where R is the number of classes and m is the number of data samples in a "batch".

$$\boldsymbol{W} = \begin{bmatrix} & & i & i+1 & & \\ 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \dots & -1 & 0 & \dots & 0 \\ \vdots & \dots & 1 & \vdots & \dots & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 \end{bmatrix} \quad \boldsymbol{K}(xi, \cdot) = \begin{bmatrix} K(x_i, x_1) \\ k(x_i, x_2) \\ \dots \\ k(x_i, x_m) \end{bmatrix}$$

$$\downarrow$$

$$\boldsymbol{W} = \begin{bmatrix} & & i & i+1 & & \\ 0 & \dots & 0 & 1 & \dots & 0 \\ \vdots & \dots & -1 & -1 & \dots & 0 \\ \vdots & \dots & \dots & \vdots & \dots & \vdots \\ 0 & \dots & 0 & 1 & \dots & 0 \end{bmatrix} \quad \boldsymbol{K}(x_{i+1}, \cdot) = \begin{bmatrix} K(x_{i+1}, x_1) \\ k(x_{i+1}, x_2) \\ \dots \\ k(x_{i+1}, x_m) \end{bmatrix}$$

We update each column of **W** after each sample t, thus, the sampled with index that is not in t will end up with zero when multiplying the non-updated elements(zeros) in the weight matrix, thus not contributing to the sum, which gives us $W \cdot K(\boldsymbol{x}_t, \cdot) = \sum_{i=0}^{t} \alpha_i K(xi, xt) =$ the confidence level for each classifier. Then, we take the classifier with the maximum confidence to give us the predicted label. If the prediction is wrong, it means that we have given too many weights on the that classifier and too few weights on the correct one, thus we increase the weight of the correct classifier and reduce the wrong one. this is represented by changing the t th column of **W**, where two rows: class y and class y', are modified. Note that the update and sample order do not need to be continuous, the not updated weights are always zero, which means that we can simulate the online environment with a random order of samples.

After implementing the above algorithm to handwritten digit classification task, we performed 20 runs for $d = 1, \dots, 7$ each run by randomly splitting zipcombo into 80% for training and 20% for test. The mean of test and train error rates as well as their standard deviations are reported in the table below:

d	Train Error Rates(%)	Test Error Rates (%)
1	8.859 ± 0.258	9.379 ± 1.449
2	2.956 ± 0.241	5.253 ± 1.057
3	1.548 ± 0.158	4.065 ± 0.591
4	1.017 ± 0.104	3.844 ± 0.466
5	0.763 ± 0.088	3.465 ± 0.538
6	0.608 ± 0.085	3.419 ± 0.447
7	0.594 ± 0.110	3.597 ± 0.415

Table 1: Mean and standard deviations of test and train error rates

From this table, we can see that the train and test error rates both decreases as d goes up

in the beginning. At around $d = 6$, the train error continues to go down as the test error fluctuates and starts to bounce up.

2) We perform the classifier with 20 runs, when using the 80% training data split from within to perform 5-fold cross-validation to select the best parameter d then retrain on full 80% training set using d and then record the test errors on the remaining 20%. The optimal d values and their test error rates are shown in table 2.

   The mean and std for optimal d: 5.75 ± 0.942

   The mean and std for test error: 3.52% ± 0.463%

optimal d	Test Error Rates (%)	optimal d	Test Error Rates (%)
7	3.333333333333333	5	4.032258064516129
5	3.225806451612903	4	3.7096774193548385
5	3.225806451612903	4	3.1720430107526884
7	3.5483870967741936	7	2.956989247311828
6	3.763440860215054	6	3.4408602150537634
5	2.903225806451613	6	4.7311827956989246
6	3.494623655913978	7	3.7096774193548385
6	3.602150537634408	7	2.903225806451613
6	3.387096774193549	5	3.0107526881720432
5	4.3010752688172046	6	3.870967741935484

Table 2: 20 optimal d and the corresponding test errors

Other Parameters: We also have a learning rate parameter for the size of each update on the weight. We believe that changing the learning rate will not make a difference as long as the convergence threshold is the same, and smaller learning rate will only result in longer convergence. Thus the learning rate parameter is not cross-validated over.

3) We performed 20 runs using the best d to find the rate of error in the test set for each digit (a, b), where a is the label and b is the wrongly predicted label. The result is shown in table 3, we can see that the model can easily distinguish between (1,0); (7, 0); (6, 3); (2, 5); (3, 6); (7, 6); (9, 6) and (6, 7), and it is harder for it to classify (6, 0); (3, 2); (8, 2); (5, 3); (8, 3); (9, 4); (3, 5); (8, 5); (9, 7); (3, 8) and (4, 9) correctly. The digit with label 3 has the highest rate to be predicted as a 5.

	0	1	2	3	4	5	6	7	8	9
0	0 ± 0	0.031 ± 0.093	0.24 ± 0.331	0.27 ± 0.253	0.175 ± 0.214	0.27 ± 0.387	0.445 ± 0.52	0.047 ± 0.113	0.237 ± 0.257	0.065 ± 0.131
1	0 ± 0	0 ± 0	0.036 ± 0.109	0.077 ± 0.154	0.333 ± 0.339	0 ± 0	0.197 ± 0.239	0.112 ± 0.256	0.313 ± 0.651	0.04 ± 0.121
2	0.395 ± 0.606	0.086 ± 0.206	0 ± 0	0.702 ± 0.58	0.516 ± 0.505	0.106 ± 0.214	0.189 ± 0.356	0.541 ± 0.5	0.411 ± 0.582	0.214 ± 0.312
3	0.364 ± 0.457	0.032 ± 0.142	1.165 ± 0.76	0 ± 0	0.054 ± 0.161	2.615 ± 1.585	0 ± 0	0.284 ± 0.418	1.103 ± 1.027	0.118 ± 0.237
4	0.059 ± 0.177	0.697 ± 0.59	0.56 ± 0.474	0.117 ± 0.305	0 ± 0	0.197 ± 0.361	0.67 ± 0.704	0.62 ± 1.054	0.15 ± 0.317	1.231 ± 0.852
5	1.105 ± 0.644	0.122 ± 0.367	0.343 ± 0.41	1.858 ± 0.998	0.364 ± 0.562	0 ± 0	0.907 ± 0.936	0.031 ± 0.135	0.828 ± 1	0.595 ± 0.789
6	1.069 ± 1.043	0.46 ± 0.582	0.335 ± 0.489	0 ± 0	0.517 ± 0.51	0.457 ± 0.579	0 ± 0	0 ± 0	0.403, 0.698	0.064 ± 0.191
7	0.0 ± 0.0	0.135 ± 0.27	0.507 ± 0.546	0.257 ± 0.373	0.859 ± 0.708	0.031 ± 0.134	0 ± 0	0 ± 0	0.291 ± 0.323	0.936 ± 0.769
8	0.821 ± 0.677	0.352 ± 0.524	1.131 ± 0.831	1.712 ± 1.139	0.595 ± 0.591	1.304 ± 0.866	0.21 ± 0.443	0.46 ± 0.546	0 ± 0	0.323 ± 0.425
9	0.122 ± 0.306	0.06 ± 0.179	0.285 ± 0.425	0.153 ± 0.265	1.004 ± 0.839	0.16 ± 0.439	0 ± 0	1.446 ± 1.296	0.152 ± 0.323	0 ± 0

Table 3: Confusion matrix of error rates on each digits in percentage, the leftmost column shows the truth and the top row is the predicted number class

4) The top five hardest to predict digit samples are shown in figure 1, this does not look surprising because they seem to be mislabeled and even human eye cannot classify them to their labeled digit.

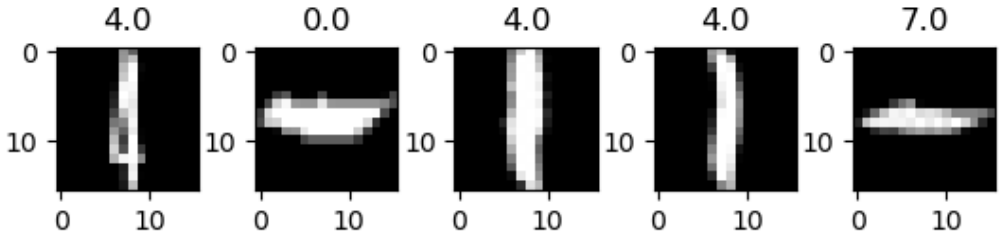

Figure 1: Top five most predicted wrong samples

5) We tried the list of c values: $[2^{-1}, 2^{-2}, \ldots, 2^{-7}]$. Adopting the trend used in the first coursework, we figured that the value of c for the Gaussian kernel should also be in a exponential decaying trend. We initially tried the kernel with base 5, and the error rate becomes similar to a random guess at 5^{-7}. After some experimentation, we found that the stretch of base 2 over the different powers gives the most appropriate results. The train and test error rate are shown in table 4.

2^x	Train Error Rates(%)	Test Error Rates (%)
-1	0.193 ± 0.079	6.065 ± 0.462
-2	0.208 ± 0.052	5.927 ± 0.502
-3	0.198 ± 0.060	5.401 ± 0.316
-4	0.269 ± 0.086	4.296 ± 0.501
-5	0.376 ± 0.090	3.250 ± 0.270
-6	0.647 ± 0.099	3.454 ± 0.601
-7	1.358 ± 0.174	3.866 ± 0.604

Table 4: Mean and standard deviations of test and train error rates for Gaussian kernel with different power of 2

According to the table, although the training error for the Gaussian kernel shows a different trend compared to the poly kernel, they have similar "optimal" test error rates in the chosen range of parameters. As c decreases exponentially, the training error increases, but the test error drops until around $c = 2^{-6}$.

Similar to section 2, We performed the classifier with 20 runs, when using the 80% training data split from within to perform 5-fold cross-validation to select the best parameter d then retrain on full 80% training set using d and then record the test errors on the remaining 20%. The cross validated optimal parameters and their corresponding test error rates are shown in table 5.

The mean and std for c(2^x): -5.75 ± 0.433

The mean and std for test error: $3.76 \pm 0.604(\%)$

optimal c 2^x	Test Error Rates (%)	optimal c 2^x	Test Error Rates (%)
-6	3.225806451612903	-6	4.3548387096774194
-5	3.064516129032258	-6	5.053763440860215
-5	3.5483870967741936	-6	4.1935483870967745
-6	4.67741935483871	-5	3.2795698924731186
-6	3.602150537634408	-6	4.7311827956989246
-6	3.978494623655914	-6	3.870967741935484
-6	4.7311827956989246	-5	3.8172043010752686
-5	3.494623655913978	-6	3.1720430107526884
-6	3.4408602150537634	-6	4.1935483870967745
-6	3.8172043010752686	-6	5.806451612903226

Table 5: 20 optimal c and the corresponding test errors

6) One-vs-One (OvO for short) is another method for using binary classification algorithms for multi-class classification. Like one-vs-rest, one-vs-one splits a multi-class classification dataset into binary classification problems. Unlike one-vs-rest, it each classifier is still a binary classifier and is constructed for each combination of two classes from the multi-class data. So there are $\frac{R\times(R-1)}{2}$ voters for R classes, thus this will have a larger complexity as R becomes larger.

Algorithm 2 OvO Kernel Perceptron

Input: $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m)\} \in (\Re^n, \{0 \ldots R\})^m$

Compute the kernel matrix $K_{ij} = K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)$ for $\forall x_i, x_j \in \mathbf{X}$
$z = R \cdot (R-1)/2$
$\mathbf{W} \leftarrow \{0\}^{z \cdot m}$
for epoch $\leq$ max_epochs **do**
  for $t \leq m$ **do**
    $\hat{y}_{vote} \leftarrow \text{sign}(W \cdot K(\cdot, \boldsymbol{x}_t))$
    Calculate the $\hat{y}$ through counting the total votes for each class
    for s from 1 to $R \cdot (R-1)/2$ **do**
      if $\hat{y} \neq y_t$ **then**
        if $W_{s,t}$ has y_t **then**
          $W_{s,t} \leftarrow 1$ if y_t is positive
          $W_{s,t} \leftarrow -1$ if y_t is negative
        end if
      end if
    end for
  end for
end for

The algorithm mostly follows that of the one verses rest algorithm. This time, the prediction is decided via a voting for each binary classifier used. The class with the most vote becomes the prediction. In addition, for miss-classified samples, we believe that all the binary classifiers that are related to the miss-classification need to be updated, i.e. the classifiers that contains the wrong predicted class and the classifiers that contains the label class. The train and test error for OvO classifier is shown in table 6.

Similar to section 2, We performed the classifier with 20 runs, when using the 80%

d	Train Error Rates(%)	Test Error Rates (%)
1	6.211 ± 0.227	7.487 ± 1.002
2	2.287 ± 0.160	4.578 ± 0.631
3	1.381 ± 0.195	3.618 ± 0.367
4	1.083 ± 0.127	3.634 ± 0.294
5	0.818 ± 0.131	3.538 ± 0.356
6	0.744 ± 0.116	3.645 ± 0.399
7	0.712 ± 0.095	3.860 ± 0.409

Table 6: Mean and standard deviations of test and train error rates for OvO classifier

training data split from within to perform 5-fold cross-validation to select the best parameter d then retrain on full 80% training set using d and then record the test errors on the remaining 20%. The optimal d values and their test error rates for the OvO classifier are shown in table 7.

The mean and std for optimal d: 4.95 ± 0.921

The mean and std for test error: $3.626\% \pm 0.515\%$

optimal d	Test Error Rates (%)	optimal d	Test Error Rates (%)
5	4.086021505376344	5	2.6881720430107527
3	4.086021505376344	5	3.602150537634408
7	3.602150537634408	6	3.0107526881720432
4	3.7096774193548385	4	3.118279569892473
5	4.4623655913978495	4	2.5806451612903226
4	3.655913978494624	6	3.870967741935484
6	3.8172043010752686	4	3.064516129032258
5	3.387096774193549	5	4.3548387096774194
5	3.387096774193549	5	3.978494623655914
5	3.870967741935484	6	4.1935483870967745

Table 7: 20 optimal d and the corresponding test errors for OvO classifier

Comparing results of the Gaussian to the polynomial Kernel: In this section, we test the Gaussian kernel, which can map the input space to infinite dimensions, instead of the polynomial kernel. Accuracy is the main comparison between these two kernels. As can be observed, the error rate of Gaussian kernel is 3.76 ± 0.604 and the polynomial kernel error rate is 3.52 ± 0.463. Under our settings, the test error rates for polynomial and Gaussian kernels are nearly the same (The polynomial one is slightly lower). The reason why Gaussian kernel didn't perform well is probably because we were unable to determine the exact optimal value for c.

2 Part 2

2.1 Semi-supervised Learning via Laplacian Interpolation

1) Two tables below contains our results of error rates and standard deviation for semi-supervised learning via Laplacian (kernel) interpolation. Table 4 shows the result for Laplacian Interpolation whereas table 5 shows the result for Laplacian Kernel Interpolation.

Mean and standard deviations of error rates(%)					
	Number of labeled data per class				
size	1	2	4	8	16
50	22.55 ± 10.54	13.85 ± 11.27	8.10 ± 5.39	5.12 ± 2.41	4.12 ± 1.71
100	5.30 ± 3.28	6.96 ± 5.04	4.32 ± 1.00	4.08 ± 0.95	2.77 ± 0.98
200	5.40 ± 5.83	3.75 ± 4.63	2.88 ± 2.74	1.98 ± 0.93	1.68 ± 0.73
400	4.68 ± 2.38	2.93 ± 2.13	2.13 ± 1.27	1.42 ± 0.59	1.09 ± 0.19

Table 8: Mean and standard deviations of error rates for Laplacian Interpolation

Mean and standard deviations of error rates(%)					
	Number of labeled data per class				
size	1	2	4	8	16
50	14.23 ± 14.90	6.66 ± 6.53	5.98 ± 3.88	4.82 ± 1.66	3.75 ± 1.51
100	3.94 ± 0.76	5.71 ± 3.08	3.75 ± 0.71	3.89 ± 0.77	2.71 ± 0.99
200	2.70 ± 2.87	2.34 ± 1.59	2.00 ± 1.38	1.67 ± 0.45	1.49 ± 0.59
400	1.54 ± 0.41	1.39 ± 0.62	1.28 ± 0.18	1.15 ± 0.27	1.00 ± 0.20

Table 9: Mean and standard deviations of error rates for Laplacian Kernel Interpolation

2) From the above table, we can see that:

a) For both models, the error rate decreases (accuracy increases) as the number of labeled data points increase, and the variance decreases.

b) For both models, the error rates decreases as the number of total data points increase.

c) The Laplacian kernel does extremely better than the original Laplacian interpolation at small $\frac{\#labeled}{\#total}$ (proportion of labeled data), the two models has nearly the similar rates when the ratio is higher.

d) Generally, both models perform really well on this task with a low error rate.

3) We see that the performance of the Laplacian kernel is better than that of the original Laplacian interpolation. This is even more obvious at low $\frac{\#labeled}{\#total}$ value. We believe that the main reason for this is that the original interpolation calculate weight only based on neighbors, thus each point can only be propagated through their three neighbors with random walk. In contrast, the Laplacian kernel captures the global information by taking the pseudo-inverse so that it incorporates distance.

4) The majority of our Laplacian Kernel Interpolation methods outperform the original Laplacian interpolation. The explanation for this is given in question 3. However, when

the proportion of labeled points reaches a particular threshold, for instance, if all unlabeled data points are related to labeled data points, the label information may be obtained directly without reduction, and the kernel version of Laplacian Interpolation would still obtain information globally. In this situation, the original Laplacian interpolation may perform better than the kernel one.

Moreover, the result displayed in our table could also provide as evidence for this speculation. It is evident that the error rates are identical for the original Laplacian Interpolation and its kernel version when the number of labeled data reaches 16, indicating that if the number of labeled data continues to increase, the original Laplacian interpolation may perform better than its kernel counterpart.

3 Part 3

3.1 Questions

a) The sample complexity (SC) for the four different approaches perceptron, winnow, least squares, and 1-nearest neighbors is illustrated in the following four images. The algorithm is explained in detail in question b. We execute this method five epochs and obtain the mean and standard deviation of sample complexity for each n. Due to the fact that the calculation for the 1-nearest neighbors is extremely time-consuming, we only test dimensions n between 1 and 15. For the remaining three algorithms, we examine dimensions between 1 and 100.

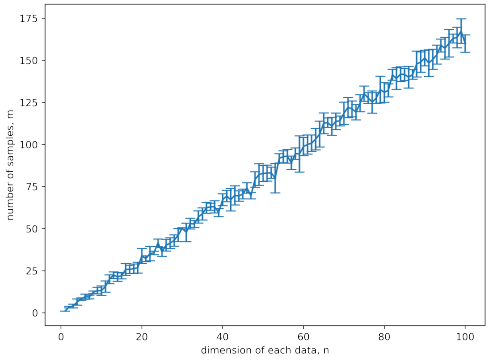

Figure 2: SC of Perceptron

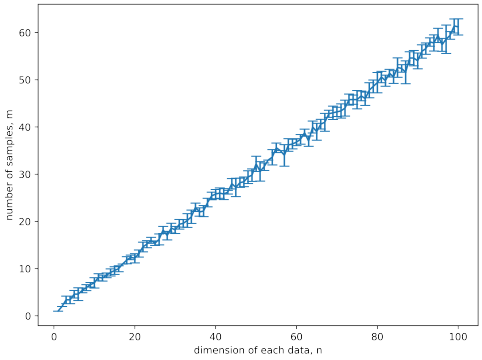

Figure 3: SC of Least Squares

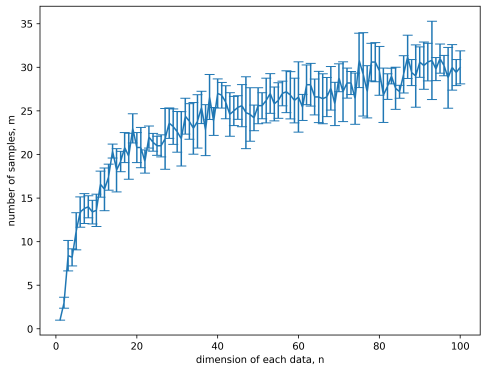

Figure 4: SC of Winnow

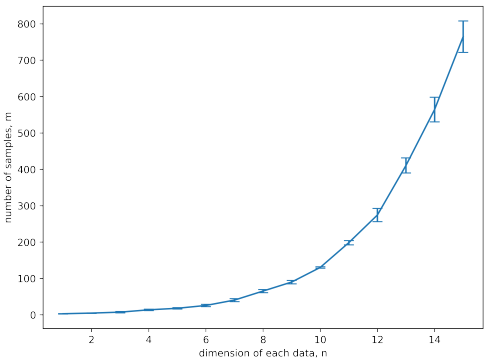

Figure 5: SC of One Nearest Neighbor

b) **Method for estimating sample complexity in details:**
The pseudo algorithm to calculate the sample complexity (SC) is shown in Algorithm 3. For our implementation, it is necessary to input n_{max} (Maximum number of dimensions for calculating the SC), s (size of test dataset), $algo$ (algorithm used for prediction) and max_{epoch} (number of runs to calculate the average and standard deviation of SC and also number of runs to calculate the average generalisation error of prediction). The first loop in this algorithm is to get the mean and standard derivation of the SC (the minimum number of examples (m) to incur no more than 10% generalisation error) for each n. This would be used to draw our graphs for the question a above as well. The next loop is to

transverse all the possible dimension n until it reaches n_{max}. Inside this loop we would find the sample complexity for current n. The way to calculate the generalisation error rate for my method in each epoch is shown below:

$$\mathcal{E}(\mathcal{A}_S) := s^{-1} \sum_{\boldsymbol{x} \in \{-1,1\}^n} I[\mathcal{A}_S(\boldsymbol{x}) \neq x_1] \tag{3.1}$$

s is the size of test dataset. For each m, we would calculate this generalisation error rate in the average of max_{epoch} times to get the sample complexity in the following way:

$$C(\mathcal{A}) := \min\Big\{ m \in \{1, 2, ...\} : \mathbb{E}\Big[\mathcal{E}\Big(\mathcal{A}_{S_m}\Big)\Big] \leq 0.1 \Big\} \tag{3.2}$$

In each epoch, we would generate the train and test datasets depending on the given s (size of test dataset) and then make predictions based on the input $algo$, as shown in lines 8 through 12 of the following algorithm: Algorithm 3.

Algorithm 3 Sample Complexity

Input: n_{max} (Max dimensions for test), max_{epoch} (number of runs to calculate the avg and std of SC and average generalisation error of prediction), s (size of test dataset), $algo$ (algorithm for prediction)

Output: Average and standard derivation of sample complexity for each n (dimension)

1: Initialise the $Complexity_{all}$ for storing all the SC (minimum m) for each n of each epoch
2: **for** epoch $\leq max_{epoch}$ **do**
3: Init $Complexity_{current}$ for storing sample complexity for each n in current epoch
4: **for** n $\leq n_{max}$ **do**
5: $m \leftarrow 0$
6: **while** average generalisation error > 0.1 **do**
7: $m \leftarrow m + 1$
8: **for** inside_epoch $\leq max_{epoch}$ **do**
9: Generate train and test dataset (based on the input s)
10: Make prediction on test dataset (based on the input $algo$)
11: Calculate the current error
12: **end for**
13: Calculate the average generalisation error
14: **end while**
15: $Complexity_{current}[n] \leftarrow m$
16: **end for**
17: $Complexity_{all}[epoch] \leftarrow Complexity_{current}$
18: **end for**
19: **return** Mean and std of the $Complexity_{all}$

The trade-offs and biases of our method:

The most obvious trade-off is between the limitation of computational and time resources and accuracy of the result. There are two points that should be considered on this subject. Firstly, we set the size of test dataset to a constant value, which is lower than 2^n the majority of the time. If test sample sizes were increased, the final result would be more precise. However, this process is also time-consuming. Secondly, to determine the mean and standard deviation of the sample complexity for each n, we execute the algorithm only 5 times. Increase the number of repeated runs could attain a more precise result with a lower standard deviation. However, this process is time-consuming as well.

The first bias of our method would be smaller estimations of sample complexity. Since our method calculates the estimates of sample complexity using a *while* loop and increases the number of samples, m until the condition, which is to incur no more than a 10% generalisation error, is reached, the sampled dataset could meet this condition earlier. The second bias would be a greater variance in sample complexity because, as described in the trade-off section, the size of the sampled dataset is constant rather than 2^n. If computational resources are available, we should use the exact size of the dataset instead of sampling to reduce possible biases.

c) According to the plots shown in the question a, there are some general observations could be found:

   * Winnow obtains the lowest sample complexity among these four algorithms, meaning it requires the smallest number of samples, n, to incur a generalisation error of no more than 10%. After the dimension n reaches 40, the rise of sample complexity becomes stable. After $n = 80$, the SC seems to have converged to approximately 30.
   * The sample complexity increases linearly for both the least squares and perceptron algorithms. However, the least squares technique needs fewer samples to satisfy the generalised error rate requirement.
   * 1-nearest neighbours requires the maximum number of examples to incur no more than a 10% error rate. As shown in the previous graph, the sample complexity of 1-NN increases exponentially. The SC reaches approximately 750 when the dimension is 15.

   To further validate these observations, a polyfit is performed on the increasing sample complexity lines indicated in question a for all four algorithms. The results of polyfit (the relationship between m and n) are shown below and we use Θ to represent this relationship:

   * Winnow: $m = 6.59 \cdot \log n + 0.05$, $\Theta(\log(n))$
   * Perceptron: $m = 1.69 \cdot n - 1.87$, $\Theta(n)$
   * Least Squares: $m = 0.60 \cdot n + 1.00$, $\Theta(n)$
   * 1-Nearest Neighbors: $m = \exp(0.41 \cdot n + 0.78)$, $\Theta(\exp(0.41)^n)$.

   Clearly, the sample complexity of the winnow algorithm grows logarithmically, whereas the 1-Nearest Neighbors algorithm grows exponentially. Then, the sample complexity for both the least squares and perceptron techniques increases linearly.

d) The upper bound of perceptron mistakes for any separable dataset is presented below based on the theorem in the lecture notes:

$$M \leq \frac{R^2}{\gamma^2} \tag{3.3}$$

   R is the radius of the hyperball that could contain every data point, whereas γ is the shortest distance between each data point in the dataset and the separable hyperplane. Due to the setting of 'just a little bit' problem where $x \in \{1, -1\}^n$, then $R = \sqrt{\sum_{i=1}^{n} x_{t,i}^2} = \sqrt{n}$. To completely separate our data set in any dimension n, we can select the hyperplane formed by coordinate axes other than the one corresponding to the first dimension. In

this situation, the shortest distance is the absolute value of the first dimension of the data, which is 1 in every instance so $\gamma^2 = 1$.

Thus, we can derive our mistake upper bound by $M \leq \left(\frac{\sqrt{n}}{\gamma}\right)^2 = \frac{n}{\gamma^2} = n$. Therefore, we could 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.

$$\hat{p}_{m,n} \leq \frac{n}{m} \tag{3.4}$$

e) First of the VC-dimension for 1-NN method would be infinite. But we could write a bound for this, which is 2^n, since this is whole permutation of data points (2^n) that could be included in the training set.

According to the theorem proposed in the work[1]:

$$m_A(\varepsilon, \delta) \geq \frac{\text{VCdim}(C) - 1}{32\varepsilon} = \Omega\left(\frac{\text{VCdim}(C)}{\varepsilon}\right) \tag{3.5}$$

When the $\varepsilon = 0.1$ and $\text{VCdim}(C) = 2^n$ in this case, we could write the lower bound of sample complexity by a function $f(n)$:

$$m_A(0.1, \delta) \geq f(n) = \frac{2^n - 1}{3.2} \tag{3.6}$$

Then we could easily proof, $m = \Omega(f(n))$:

$$m = \Omega(f(n)) = \Omega(10 \cdot (2^n - 1)) \tag{3.7}$$

References

[1] Andrzej Ehrenfeucht, David Haussler, Michael Kearns, and Leslie Valiant. A general lower bound on the number of examples needed for learning. *Information and Computation*, 82(3):247–261, 1989.