# Supervised learning cw2

Qiang Hu 22063614      Houkang Zhang 22084226

December 13, 2022

## 1 Part1 Kernel perceptron

### 1.1 Set up

#### 1.1.1 Kernel

In this experiment, a kernel is added to map the data to a higher dimension space, thus it becomes separable for perceptron learning algorithm.

$$K(p,q) = (p \cdot q)^d$$

#### 1.1.2 online learning

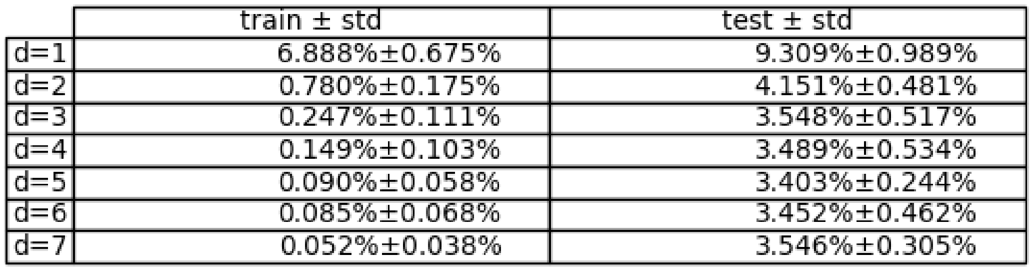
For a set of input $D = \{(x_1, y_1)....(x_n, y_n)\}$, the perceptron algorithm runs a single $(x_n, y_n)$ once a time, and repeat testing on this dataset D for fixed epochs. In this experiment, the number of epochs for the learning algorithm will not be tested over cross-validation, for the optimized test result, we run every perceptron algorithm with online learning till the algorithm converge.

#### 1.1.3 Generalizing to k class

**Algorithm 1** One-vs-Rest Perceptron algorithm

```
$n \leftarrow$ number of classes
$k \leftarrow K(X, X)$                          ▷ initialise kernel function with train data X
$W \leftarrow zeros$(number of class, data length)                          ▷ initialize weight
for 1 to $p$ do                          ▷ maximum epoch
    $Error \leftarrow 0$                          ▷ total error in prediction in one epoch
    for 1 to $j$ do                          ▷ number of datapoints
        $y_i = \sum_{i=0}^{i=n}(w_{i,j} \cdot k_{i,j})$                          ▷ predict the class $y_i$ by kernel function
        $y_{pred} \leftarrow$ index of Maximum in $y_i$    ▷ the predicted value is the index of the maximum value
        if $y_{pred} \neq y_{real}$ then Error ++
        end if
        $w_{y_{pred},i} \leftarrow w_{y_{pred},i} - 0.5 * |y_{pred} - y_{real}|$ ▷ update the weight by the different of predict value and real value
        $w_{y_{real},i} \leftarrow w_{y_{pred},i} - 0.5 * |y_{pred} - y_{real}|$ ▷ update the weight by the different of predict value and real value
    end for
    if Error in epoch p = Error in epoch (p-1) then break                          ▷ algorithm converge and stop
    end if
end for
```

One vs rest algorithm maintains a weight of the whole data sample, when predicting input data, it compares the input data weight with the weight of the whole dataset, and the most likely value is the predicted value. The $y_i$ is a set of magnitudes of each index(n classes) prediction made by kernel matrix, the index with the maximum magnitude is the predicted value, and we update the weight by half of the absolute difference between the real value and the predicted value thus the weight will converge to the real value after a number of iteration.



| | train ± std | test ± std |
|---|---|---|
| d=1 | 6.888%±0.675% | 9.309%±0.989% |
| d=2 | 0.780%±0.175% | 4.151%±0.481% |
| d=3 | 0.247%±0.111% | 3.548%±0.517% |
| d=4 | 0.149%±0.103% | 3.489%±0.534% |
| d=5 | 0.090%±0.058% | 3.403%±0.244% |
| d=6 | 0.085%±0.068% | 3.452%±0.462% |
| d=7 | 0.052%±0.038% | 3.546%±0.305% |

Figure 1: Basic results of training error and testing error with std

#### 1.1.4 Cross validations

split the dataset into five folds and test using one fold as test data and the rest as training data alternatively, and calculate the mean error.

## 1.2 Questions

#### 1.2.1 Question 1: Basic results

Run the algorithm from $d = 1....7$, with the maximum epoch of 20, note most of the epochs can converge at the 8th epoch on average. And less bad than 18 depends on d. As seen in figure 1 both the train and the test error is decreasing with the increase of d, but when d gets larger than 5, the test error is getting larger at d = 6 thus the best d is between d = 4 to d = 6.

#### 1.2.2 Question 2: Cross validation

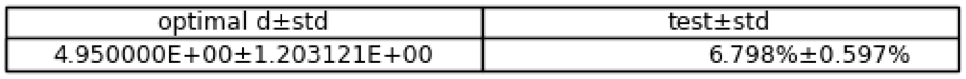
Split 80% of the data to perform cross-validation to determine the best d and use that d on the rest 20% data to calculate the testing error. See figure 2 as in question one the optimal $d = 4.95 \approx 5$.

#### 1.2.3 Question 3: Confusion matrix

Use cross-validation in question2 to generate a confusion matrix, matrix[x,y] is the probability of predicting the correct value x as y. See Figure 3, the algorithm is most like to predict 3 incorrectly into 5 and vice versa and as for predicting 4 and 9, 7 and 9, 5 and 8 also generate relatively high errors, and the mistakes of predicting those values are reasonable because they are sometimes confusing in identification in handwriting.

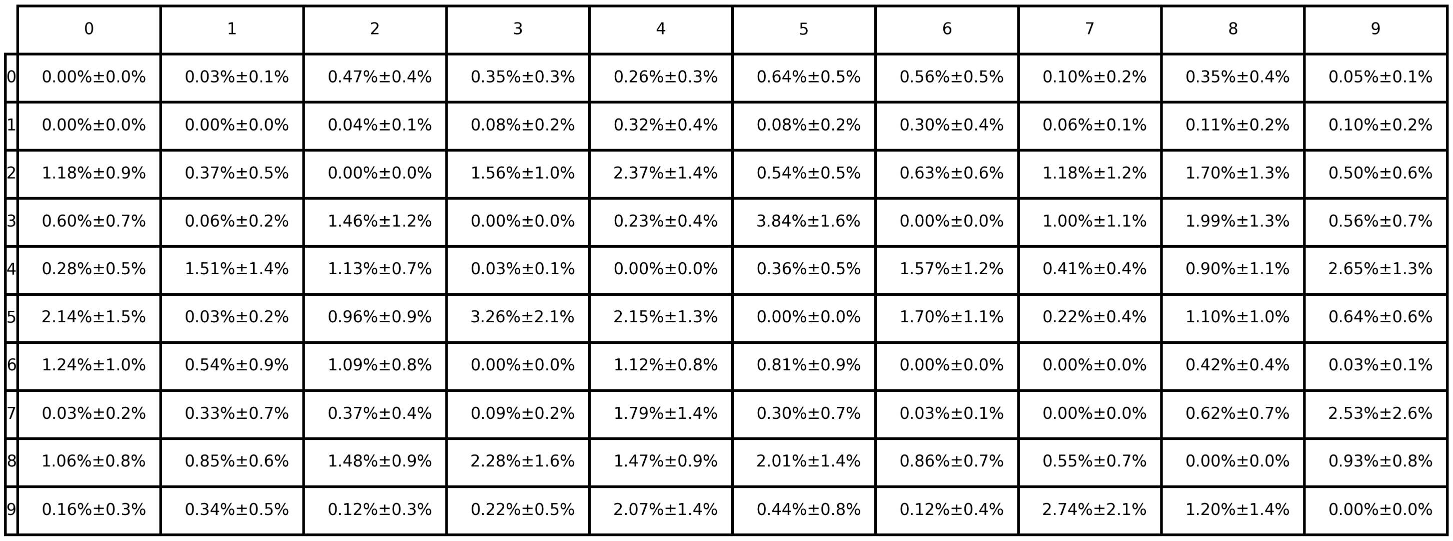
#### 1.2.4 Question 4: Most confusing images

Run a large number of iterations and count the number of predicting an image incorrectly, and take the top five images with the largest counting number. See Figure 4, some of the images are ridiculous to identify.



| optimal d±std | test±std |
|---|---|
| 4.950000E+00±1.203121E+00 | 6.798%±0.597% |

Figure 2: train error and test error by cross validation



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.00%±0.0% | 0.03%±0.1% | 0.47%±0.4% | 0.35%±0.3% | 0.26%±0.3% | 0.64%±0.5% | 0.56%±0.5% | 0.10%±0.2% | 0.35%±0.4% | 0.05%±0.1% |
| 1 | 0.00%±0.0% | 0.00%±0.0% | 0.04%±0.1% | 0.08%±0.2% | 0.32%±0.4% | 0.08%±0.2% | 0.30%±0.4% | 0.06%±0.1% | 0.11%±0.2% | 0.10%±0.2% |
| 2 | 1.18%±0.9% | 0.37%±0.5% | 0.00%±0.0% | 1.56%±1.0% | 2.37%±1.4% | 0.54%±0.5% | 0.63%±0.6% | 1.18%±1.2% | 1.70%±1.3% | 0.50%±0.6% |
| 3 | 0.60%±0.7% | 0.06%±0.2% | 1.46%±1.2% | 0.00%±0.0% | 0.23%±0.4% | 3.84%±1.6% | 0.00%±0.0% | 1.00%±1.1% | 1.99%±1.3% | 0.56%±0.7% |
| 4 | 0.28%±0.5% | 1.51%±1.4% | 1.13%±0.7% | 0.03%±0.1% | 0.00%±0.0% | 0.36%±0.5% | 1.57%±1.2% | 0.41%±0.4% | 0.90%±1.1% | 2.65%±1.3% |
| 5 | 2.14%±1.5% | 0.03%±0.2% | 0.96%±0.9% | 3.26%±2.1% | 2.15%±1.3% | 0.00%±0.0% | 1.70%±1.1% | 0.22%±0.4% | 1.10%±1.0% | 0.64%±0.6% |
| 6 | 1.24%±1.0% | 0.54%±0.9% | 1.09%±0.8% | 0.00%±0.0% | 1.12%±0.8% | 0.81%±0.9% | 0.00%±0.0% | 0.00%±0.0% | 0.42%±0.4% | 0.03%±0.1% |
| 7 | 0.03%±0.2% | 0.33%±0.7% | 0.37%±0.4% | 0.09%±0.2% | 1.79%±1.4% | 0.30%±0.7% | 0.03%±0.1% | 0.00%±0.0% | 0.62%±0.7% | 2.53%±2.6% |
| 8 | 1.06%±0.8% | 0.85%±0.6% | 1.48%±0.9% | 2.28%±1.6% | 1.47%±0.9% | 2.01%±1.4% | 0.86%±0.7% | 0.55%±0.7% | 0.00%±0.0% | 0.93%±0.8% |
| 9 | 0.16%±0.3% | 0.34%±0.5% | 0.12%±0.3% | 0.22%±0.5% | 2.07%±1.4% | 0.44%±0.8% | 0.12%±0.4% | 2.74%±2.1% | 1.20%±1.4% | 0.00%±0.0% |

Figure 3: Confusion matrix

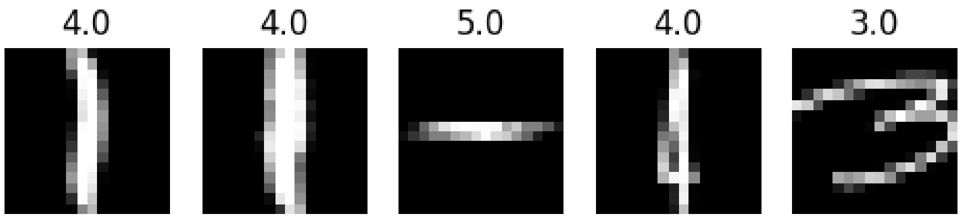


Figure 4: Top five confusing images

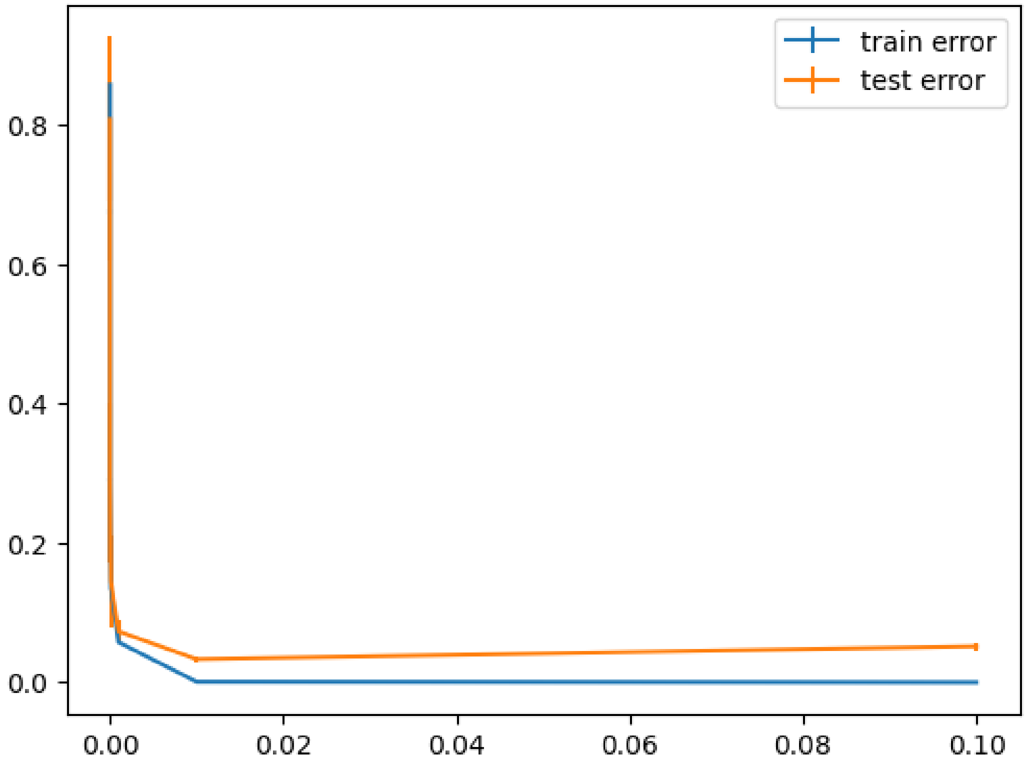


Figure 5: $c = 10^{-d}$ with S = 1....7

### 1.2.5 Question 5: Gaussian kernel

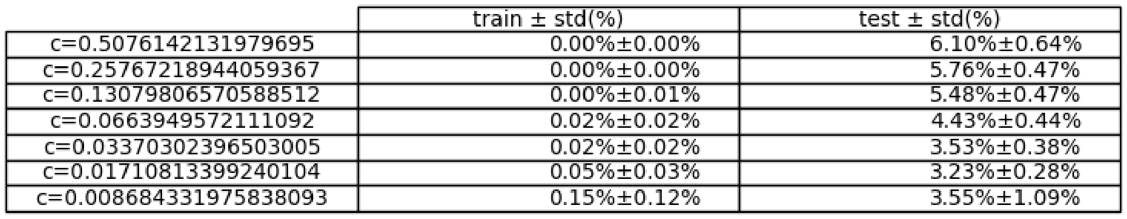
Extend the kernel into the Gaussian kernel:

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

as

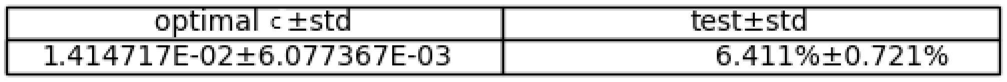
$$c = \frac{1}{2\sigma^2}$$

c should be valued lies between 0 and 1 and exponentially declining. Perform a decline on the scale of $10^{-d}$ with d in S = 1....7 against train and test error rate to determine S Figure 5, it shows that the minimum test error appears at around c = 0.01, in order to obtain a clear result, we choose $c = 1.97^{-d}$ with d in S = 1....7 in next experiment. And repeat questions 1&2 with the Gaussian kernel with determined c in S, see Figure 6 for the basic result and Figure 7 for cross-validation. The lowest test error appears between d = 5 and d = 7 as $c = 10^{-d}$. See Figure 7 the optimal c value and test error of Gaussian kernel, comparing to the Figure 1 and 2, Gaussian kernel performed better than using polynomial kernel. I think one of the reasons for that is the Gaussian kernel calculated euclidean distance between matrices, it improved the kernel to construct more exact decision boundaries.



| | train ± std(%) | test ± std(%) |
|---|---|---|
| c=0.5076142131979695 | 0.00%±0.00% | 6.10%±0.64% |
| c=0.25767218944059367 | 0.00%±0.00% | 5.76%±0.47% |
| c=0.13079806570588512 | 0.00%±0.01% | 5.48%±0.47% |
| c=0.0663949572111092 | 0.02%±0.02% | 4.43%±0.44% |
| c=0.03370302396503005 | 0.02%±0.02% | 3.53%±0.38% |
| c=0.01710813399240104 | 0.05%±0.03% | 3.23%±0.28% |
| c=0.008684331975838093 | 0.15%±0.12% | 3.55%±1.09% |

Figure 6: Basic results for Gaussian kernel



| optimal c ±std | test±std |
|---|---|
| 1.414717E-02±6.077367E-03 | 6.411%±0.721% |

Figure 7: cross validation for Gaussian kernel

### 1.2.6 Question 6: One vs One perceptron

**Algorithm 2** One vs One perceptron algorithm

```
input data: X:train data,Y:train target
K ← K(X,X)                                                    ▷ kernel map the data to higher dimension
n ← class number
C ← n(n-1)/2 classifiers $c_i$                                 ▷ binary classifiers
for 1 to $p$ do                                               ▷ number of epoches
    Error
    for 1 to $j$ do                                           ▷ number of data points
        $confidence = sign(C \cdot k_i)$                      ▷ votes from classifier $c_i$ to $c_j$ to this data poiont
        vote                                                  ▷ store the total votes for each class
        for 1 to $n$ do $vote_n$ ← sum($confidence_n$[vote for n])   ▷ class n get a vote if the classifier
vote to it
        end for
        $y_{real} = Y_j$
        if $y_{pred} \neq y_{real}$ then
            Error++
        end if
        $y_{predict}$ ← n : $vote_n$ = max(vote)              ▷ the class with highest vote wins
        for $con_i$ in confidence do                          ▷ adjust each classifier depend on real y
            $con_i$ is the confidence of classifier $c_i$ for class a,b
            vote($c_i$) = a If $con_i \leq 0.5$ else b
            if vote($c_i$) = a = $y_{real}$ then              ▷ update $c_i$ make it tends to a $c_i$ -= learning rate
            end if
            if vote($c_i$) = b = $y_{real}$ then              ▷ update $c_i$ make it tends to b $c_i$ += learning rate
            end if
        end for
    end for
    if Error = $Error_{p-1}$ then break                       ▷ break when converge
    end if
end for
```

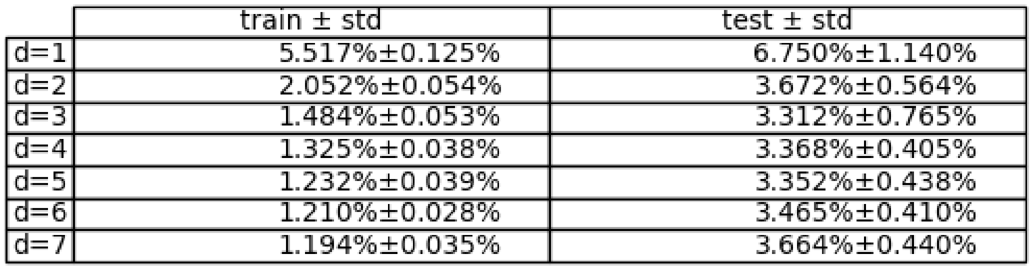
The alternative method to extend the algorithm to determine the k-class is one vs one perceptron algorithm, in one vs one algorithm we maintain a set of classifiers of the size of n(n-1)/2 and each classifier only classifies between two classes. At each training round, each classifier predicts one class label and the class label with the most votes is the predicted target value. And the algorithm evaluates each classifier with the real target value, every classifier corrected and voted with the real target value will be increased/decreased to more fit the real target. Then perform Q1&Q2 with one vs one perceptron algorithm with the polynomial kernel, see Figure 8 and 9 , one vs one performed better when d is small, but when d gets large the performance of OvO is worse that OvR method and it takes longer to run for the same dataset because one vs one need to update every classifier in each round.

## 2 Part2

(1) Plot Figure 10 for Laplacian Interpolation and Figure 11 for Laplacian Kernel Interpolation.

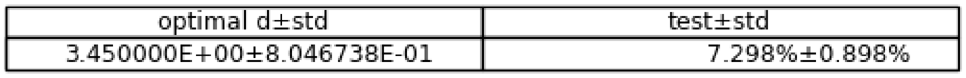
(2) For both LI and LKI, for each fixed data size when the number of known labels increases, the error rate decreases but with a decreasing speed, and the error rate of LKI finally converges somewhere but LI keeps decreasing. For example, when m = 100, and when the number of known labels L increases form 4 to 16, the error rate for LI is still decreasing where from 0.04 to 0.031. But the error rate of LKI just stays at 0.034. For each fixed number of known labels when the data size increases, the error rate also decreases and the descent is getting slower.

(3) LKI is significantly performed better than LI for each data size m when the number of known labels are small enough where $L < 8$. But when L is huge enough, which means when L = 8,



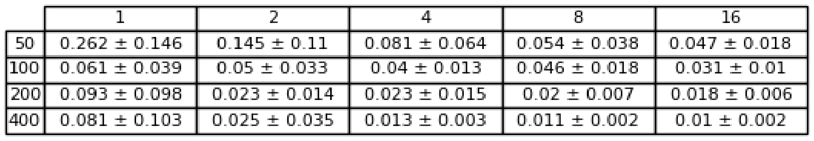
| | train ± std | test ± std |
|---|---|---|
| d=1 | 5.517%±0.125% | 6.750%±1.140% |
| d=2 | 2.052%±0.054% | 3.672%±0.564% |
| d=3 | 1.484%±0.053% | 3.312%±0.765% |
| d=4 | 1.325%±0.038% | 3.368%±0.405% |
| d=5 | 1.232%±0.039% | 3.352%±0.438% |
| d=6 | 1.210%±0.028% | 3.465%±0.410% |
| d=7 | 1.194%±0.035% | 3.664%±0.440% |

Figure 8: basic result running on one vs one perceptron



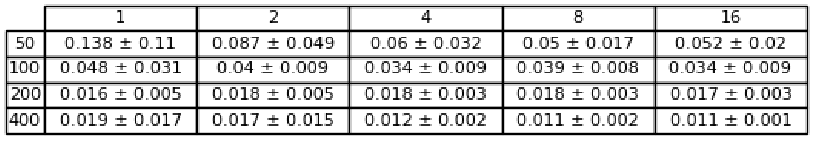
| optimal d±std | test±std |
|---|---|
| 3.450000E+00±8.046738E-01 | 7.298%±0.898% |

Figure 9: cross validation for one vs one perceptron



| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 50 | 0.262 ± 0.146 | 0.145 ± 0.11 | 0.081 ± 0.064 | 0.054 ± 0.038 | 0.047 ± 0.018 |
| 100 | 0.061 ± 0.039 | 0.05 ± 0.033 | 0.04 ± 0.013 | 0.046 ± 0.018 | 0.031 ± 0.01 |
| 200 | 0.093 ± 0.098 | 0.023 ± 0.014 | 0.023 ± 0.015 | 0.02 ± 0.007 | 0.018 ± 0.006 |
| 400 | 0.081 ± 0.103 | 0.025 ± 0.035 | 0.013 ± 0.003 | 0.011 ± 0.002 | 0.01 ± 0.002 |

Figure 10: Table for Laplacian Interpolation, where row header is known labels (per class), column header is data points per label.



| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 50 | 0.138 ± 0.11 | 0.087 ± 0.049 | 0.06 ± 0.032 | 0.05 ± 0.017 | 0.052 ± 0.02 |
| 100 | 0.048 ± 0.031 | 0.04 ± 0.009 | 0.034 ± 0.009 | 0.039 ± 0.008 | 0.034 ± 0.009 |
| 200 | 0.016 ± 0.005 | 0.018 ± 0.005 | 0.018 ± 0.003 | 0.018 ± 0.003 | 0.017 ± 0.003 |
| 400 | 0.019 ± 0.017 | 0.017 ± 0.015 | 0.012 ± 0.002 | 0.011 ± 0.002 | 0.011 ± 0.001 |

Figure 11: Table for Laplacian Kernel Interpolation, where row header is known labels (per class), column header is data points per label.

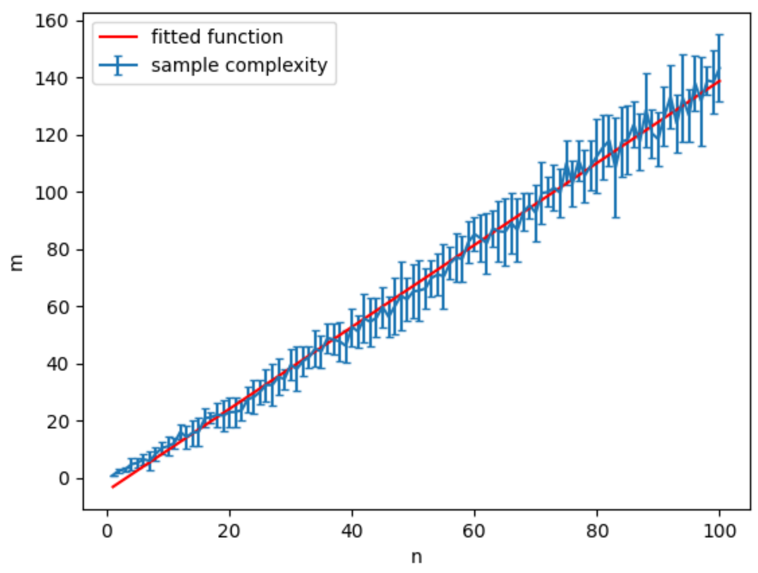


Figure 12: Estimated sample complexity for perception, where the plot with red color is fitted function.

their performances are nearly the same, and when L = 16, the performance of LI is slightly better than LKI. The reason is that LKI will be overfitted with a large L, but the performance of LI is monotonically increasing with L. Thus the error rate of LI would be small than LKI when L increases to somewhere.

(4) For the difference between the performance of LI and LKI, one of the reasons is that the number of known labels L in the dataset is not big enough. Thus when we increase L to 16 or even bigger, LI would perform better than LKI. The reason is that for LKI, when we have enough labels, the error rate will converges to somewhere (not 0), such as when m = 400, it stays at 0.011 and the std becomes quite small, but for LI, the performance will be better than itself all the time as L increases and finally converges to 0. Therefore, LI will perform better than LKI with when L is big enough.

## 3 Part3

(a) Plot Figure 12 for perceptron algorithm, Figure 13 for winnow algorithm, Figure 14 for least squares algorithm and Figure 15 for 1-nearest neighbours algorithm.

(b)

1) Firstly, we set a maximum (1500) for the rows m of the dataset, where $m \in [0, 1500]$. Then we generate randomly the test dataset for each $n \in [1, 100]$ with fixed s=maximum and training dataset for each n, but with iterable value $m_i$ whose initial value is 1. After generating those datasets, we perform the corresponding algorithm to predict $y_p$. And once we find the generalization error $\varepsilon(A_s) := \frac{1}{s} \sum_{x\in\{-1,1\}^n} I\left[A_s(x) \neq x_1\right] \leq 0.1$ (s is the test data size), we store $m_i$ as sample complexity for corresponding n. Then we repeat the above steps ten times in order to calculate the mean and std of m for each $n \in [1, 100]$.

2) Tradeoff and bias: Firstly, for the test dataset, we only generate the fixed number of samples where s=1500 instead of $2^n$. This is because when n increases, $2^n$ will be extremely large, and it would be expensive computationally if we keep using $2^n$ as our test size. Thus the result would lose some of the accuracies but decrease time complexity. Secondly, we only repeat all the steps 10 times to calculate the mean and std of sample complexity, which would also lose some of the accuracies but decrease time complexity. Thirdly, for 1-nn, we only plot part of the

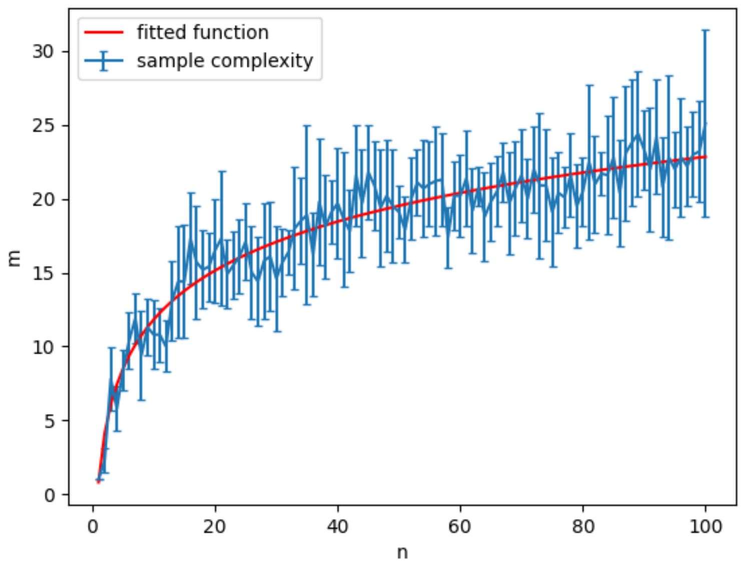


Figure 13: Estimated sample complexity for winnow, where the plot with red color is fitted function.

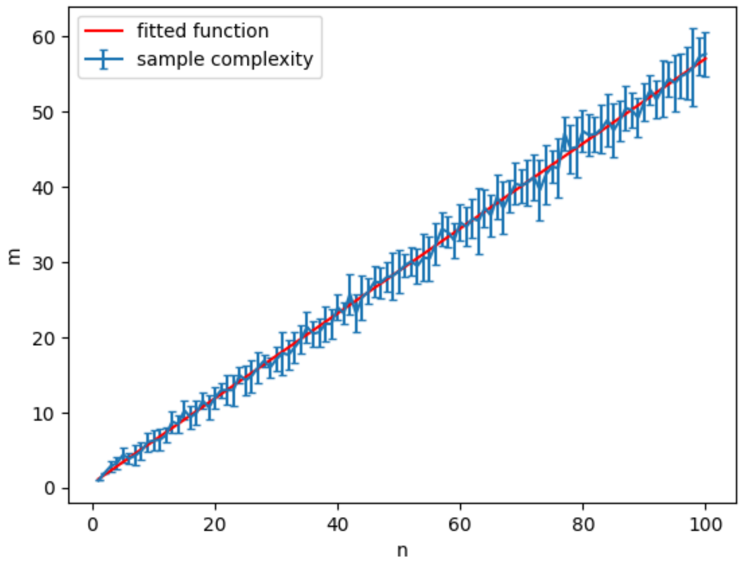


Figure 14: Estimated sample complexity for least squares, where the plot with red color is fitted function.

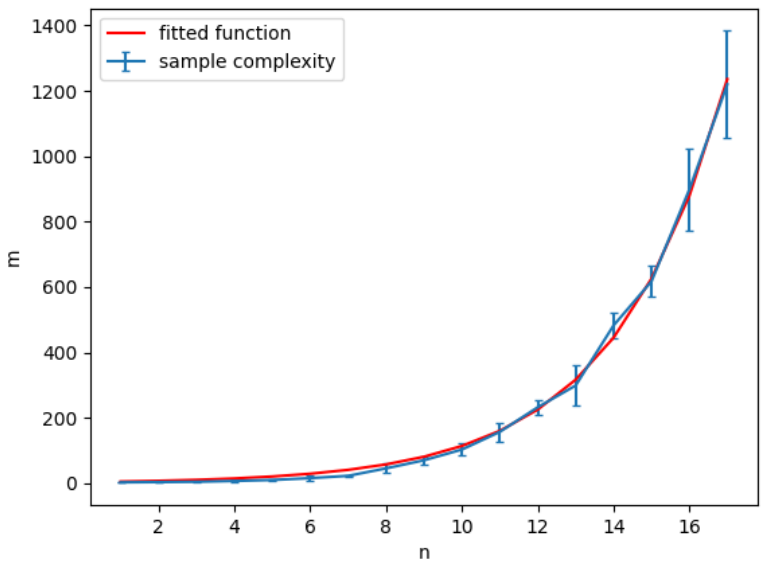


Figure 15: Estimated sample complexity for 1-nn, where the plot with red color is fitted function.

figure, where $n \in [1, 17]$, and use those data to predict its behavior as n becomes large. This is because m increases exponentially as n increases. we only plot the figure when $n \in [1, 17]$ to save computation time. Finally, for the bias, we generate all the datasets randomly, thus if we perform different algorithms on different datasets, our method would generate bias.

(c)

1) For perceptron, from Figure 12, we observe this function grows linearly, where $m = \Theta(n)$. Thus using the data we collected before, we get the fitted function: $m = 1.431n$ as $n \to \infty$.

2) For winnow, from Figure 13, we observe this function grows logarithmically, where $m = \Theta(log(n))$. Thus using the data we collected before, we get the fitted function: $m = 4.937ln(n)$ as $n \to \infty$.

3) For least squares, from Figure 14, we observe this function grows linearly, where $m = \Theta(n)$. Thus using the data we collected before, we get the fitted function: $m = 0.570n$ as $n \to \infty$.

4) For 1-nn, from Figure 15, we observe this function grows exponentially, where $m = \Theta(b^n)$ ($b = 1.41$ after calculation). Thus using the data we collected before, we get the fitted function: $m = 3.44 \times 1.41^n$ as $n \to \infty$.

5) For the performance, we observe winnow has the best performance because the growth of sample complexity is getting slower. On the contrary, 1-nn has the worst performance because the growth of sample complexity is getting faster. The growth of other two algorithms are both linear but with different ratio, where least squares has the better performance than perceptron because its ratio is smaller. Therefore, we have the rank for the performance of those algorithms when n is big enough: $winnow > leastsquares > perceptron > 1 - nn$.

(d) By Perceptron Bound Theorem (Novikoff), we know that:

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

where M is the mistakes of this algorithm, $R := max_t \|x_t\|$, and $(v \cdot x_t)y_t \geq \gamma$ for all t. Then because $x_t \in \{-1, 1\}$, we have

$$max_t \|x_t\| = \sqrt{\sum_{i=1}^{n} x_{t,i}^2} = \sqrt{n}$$

Thus $R = \sqrt{n}$. And because all $x_t, y_t \in \{-1, 1\}$. We have maximized $\gamma$ where $2 \cdot \gamma = |-1 - 1| = 2$, then $\gamma = 1$. Which means

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

And by Batch Bound Theorem, we have

$$Prob(A_s(x_s) \neq y_s) \leq \frac{B}{m}$$

where B is a mistake bound for algorithm A for any such set. We also know that sample $(x_s, y_s)$ is i.i.d which is independent of the s-1 samples. Then

$$Prob(A_s(x_s) \neq y_s) \leq \frac{M}{m} \leq \frac{n}{m}$$

Therefore,

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

(e)