Supervised Learning Assignment 2

1 Part I: Kernel perceptron (Handwritten Digit Classification)

1.1 The data

The dataset used for this Part I is found in zipcombo.dat. It consists of 9298 records (lines) each record (line) contains 257 values, the first value is the digit, the remaining 256 values represent a 16×16 matrix of grey values scaled between -1 and 1. The digits in the dataset are 0,1,2,3,4,5,6,7,8 and 9 with the frequencies of each digit shown in Figure 1. The aim of Part 1 of the assignment is to train a classifier to recognize the handwritten digits. In this exercise we will generalise the standard perceptron in two ways, first we generalize the perceptron to use kernel functions so that we may generate a nonlinear separating surface and second, we generalize the perceptron into a majority network of perceptrons so that instead of separating only two classes we may separate k classes. The training and testing sets will be a 80%-20% split.

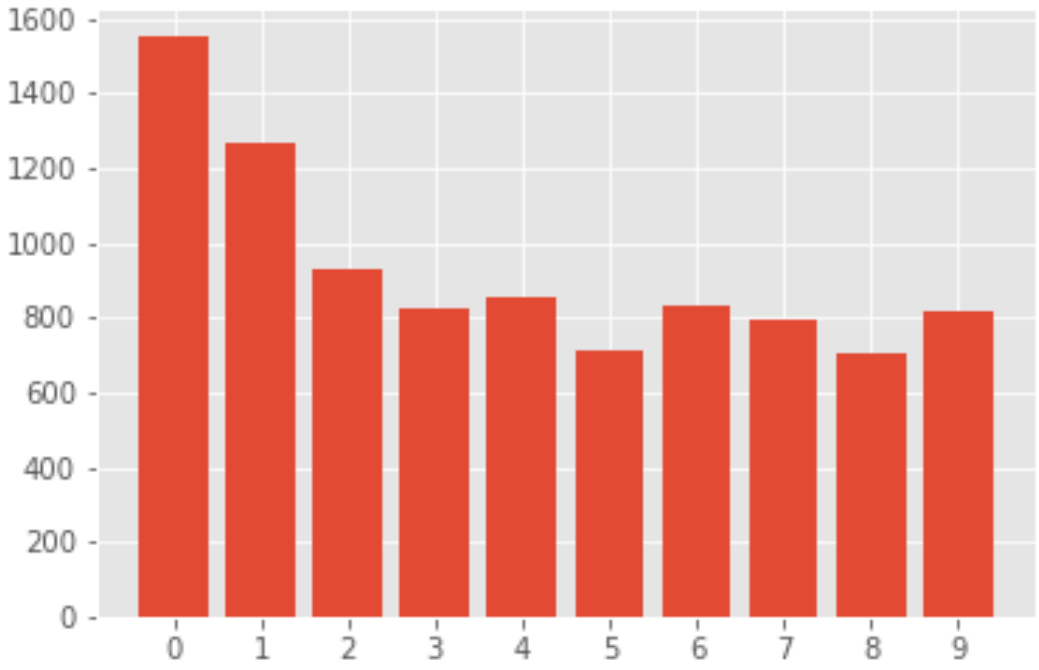

Figure 1: Frequencies of each digit in dataset.

1.2 Two Class Kernel Perceptron

We begin from the standard two class kernel perceptron.

Algorithm 1 Two class kernel perceptron (training)

Input $(\mathbf{x}_1, \mathrm{y}_1), \ldots, (\mathbf{x}_m, \mathrm{y}_m) \in R^n \times \{-1, 1\}$

Initialise $w_1 = \vec{0}$; $\alpha_0 = 0$
for $t = 1$ to m **do**
    Receive pattern: $\mathbf{x}_t \in \mathrm{R}^n$
    Predict: $\hat{\mathrm{y}}_t = \mathrm{sign}(\mathbf{w}_t(\mathbf{x}_t)) = \mathrm{sign}(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$
    Receive label: y_t
    if $\hat{\mathrm{y}}_t = \mathrm{y}_t$ **then**
        $\alpha_t = 0$
    else
        $\alpha_t = \mathrm{y}_t$
    end if
    Update $\mathbf{w}_{t+1}(\cdot) = \mathbf{w}_t(\cdot) + \alpha_t K(\mathbf{x}_t, \cdot)$
end for

1.3 Generalizing to k classes

We will generalise the 2-class kernel perceptron to k classes in two ways. The first one being One Versus Rest classification and the second one being One versus one classification.

1.3.1 OvR: One-vs.-rest

In One-vs.-rest classification we train k binary classifiers; one for each digit class in our dataset. The ith classifier (for i=0 to 9) will be trained to distinguish between digits belonging to class i vs digits belonging to any other class. To do this when training the ith classifier, we will transform data labels into -1,1 where 1 is for class i and -1 is rest. We will train the k classifiers in parallel.

Algorithm 2 OVR multi-class kernel perceptron (training) for k-classes

Input $(\mathbf{x}_1, \mathrm{y}_1), \ldots, (\mathbf{x}_m, \mathrm{y}_m) \in R^n \times \{-1, 1\}$

1: Compute the m by k matrix kclasstrain of transformed labels. Where each row represents a different training point. If the row's true label is digit j, then for that given row all the columns will contain a -1 except at the jth column where it will contain a +1.
2: compute Gram matrix K associated with the kernel function $K_{i,j} = K(x_i, x_j)$ for i,j=1 to m.
3: Initialise k by m matrix $\alpha = 0$
4: **for** *epoch* = 1 to *maxepochs* **do**
5: **for** *class* = 0 to k **do**
6: **for** $t = 1$ to m **do**
7: y_t = kclasstrain[t,k] #receive label in OVR form
8: Predict $\hat{y} = sign(matmul(\alpha[class, :], K[:, [t]]))$
9: Update α[class,t] = α[class,t] + y_t*$((\hat{y} \neq y_t))$ #update alpha only if prediction wrong
10: **end for**
11: **end for**
12: **end for**
13: Return α

Once we have the k by m matrix α we can predict the label of a new point $x^* \in R^n$ by the following:

1. Compute the m by 1 matrix $K_{i,j} = K(x_i, x^*)$ for i=1 to m (i.e. the set we trained on).

2. Compute the k by 1 confidence vector $\kappa = \alpha K$

3. Predict the label of x^* as $\mathrm{argmax}_{i \in 1,2,,\ldots,k}\{\kappa_i\}$.
   The testing process is simple. We pass each of our testing points and using the above scheme we predict their labels. Then we just count how many of the labels we predicted incorrectly out of all the testing points. This then gives us our testing error. We can similarly find the training error by simply passing our training points to the prediction scheme instead.
   In Algorithm 1 we can see that the update rule is based on the sum $w(\cdot) = \sum_{i=0}^{m} \alpha_i K(x_i, \cdot)$. In my code implementation I got rid of the $w(\cdot)$ term. Instead in Algorithm 2, when training the OVR binary classifier for digit 'class', I store each α_i(for i=0 to m) all in one row (row number 'class') of the k my m matrix α. This means that to evaluate the sum all I have to do is matrix multiply α and K. In line 8 of algorithm 2, we can also see that $matmul(\alpha[class, :], K[:, [t]]))$ uses all the m columns of α and all the m rows of K. Usually when training on datapoint 't', we only need the first 't-1' rows of α. However since, α is initialised to all 0s, the extra rows (the ones after the first 't-1') are all 0 anyway so they do not add anything to the matrix multiplication (This is true while we're in the first epoch only). If we train for more than 1 epoch, then since we are not resetting α to 0, the above statement is no longer true as the extra rows aren't 0s anymore. However, this is intentional as when we train for more than one epoch we are just cycling through the training data from the beginning. If we train for e epochs, we go through e*m

datapoints however only m of the datapoints are unique. Thus instead of making α k by e*m, I kept it k by m and added the update associated with the repeated datapoints $x_{i+m} = x_{i+2m} = ... = x_{i+e*m}$(for i=1 to m) to the ith column of alpha. This logic and represenation will carry through later when we implement the One Versus One classifiers too.

Experimental protocol

Basic results using Polynomial Kernel $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q})^d$

Here I performed 20 runs for d=1,...,7. Each run randomly splits the data into 80% train and 20% test. In Table 1 I report the mean test and train error rates as well as well as standard deviations. Here I trained for 2 epochs, as I'm training the OVR classifiers in parallel, each epoch consists of all 10 classifiers being trained on all training points. Thus to avoid possible overfitting and reduce training time, I chose 2 epochs so that each datapoint is seen and trained on at least twice.

d=1	d=2	d=3	d=4	d=5	d=6	d=7
8.97±1.4	2.12±0.55	0.79±0.13	0.49±0.13	0.38±0.25	0.24±0.09	0.21±0.06
10.52±1.45	4.82±0.64	3.6±0.39	3.5±0.52	3.21±0.41	3.18±0.4	3.19±0.46

Table 1: mean±std. Row 2 is training error, Row 3 is testing error.

As we can see, the increasing kernel power reduces the training error immensely however after d=6 it seems our testing error starts increasing. This indicates that there might be some overfitting with the higher d values as our kernel perceptron 'learns' the training data too closely and starts learning the noise in the data too. Our best test error seems to hover around 3.2%.

Cross Validation

Next we perform 20 runs under the same conditions as above however this time, we take the training data and split it into 5-folds to do 5-fold cross validation and find the best parameter d^*. In Table 2 , we return the optimal value found in each of the 20 runs with their associated test errors.

run	optimal d	test error in %
1	7	3.228
2	7	3.55
3	6	2.905
4	7	3.873
5	7	3.228
6	6	2.582
7	6	3.12
8	6	3.497
9	7	2.636
10	7	4.196
11	5	3.981
12	7	2.259
13	6	3.443
14	7	2.797
15	7	3.497
16	7	2.69
17	7	2.743
18	7	3.443
19	6	3.443
20	7	3.228

Table 2:

Here is the 3.217±0.492(= mean test error±std) and 6.6±0.583(= mean d^* with std) for the above table.

Confusion Matrix

We also return the confusion matrix associated with the optimal d values found in the cross validation above in Table 3.

	0	1	2	3	4	5	6	7	8	9
0	0.0000± 0.0000	0.1129± 0.2349	0.1991± 0.2930	0.2627± 0.3588	0.1282± 0.1840	0.0979± 0.1816	0.3870± 0.4749	0.1126± 0.1848	0.1795± 0.2133	0.0485± 0.1156
1	0.0213± 0.0927	0.0000± 0.0000	0.0793± 0.1587	0.0593± 0.1411	0.2976± 0.3469	0.0611± 0.1964	0.2435± 0.3567	0.0401± 0.1205	0.1216± 0.2296	0.0793± 0.1587
2	0.3238± 0.4030	0.2695± 0.3137	0.0000± 0.0000	0.5032± 0.7272	0.6440± 0.5495	0.2382± 0.3142	0.2378± 0.3555	0.6681± 0.6126	0.6648± 0.7653	0.0269± 0.1172
3	0.3876± 0.5069	0.3671± 0.5143	0.6400± 0.5551	0.0000± 0.0000	0.1502± 0.3175	1.8889± 1.3175	0.0584± 0.1753	0.2202± 0.3753	1.0618± 0.8854	0.0627± 0.1884
4	0.0291± 0.1267	0.5660± 0.6531	0.7487± 0.5829	0.0598± 0.1794	0.0000± 0.0000	0.1419± 0.3037	0.4931± 0.4650	0.3522± 0.3865	0.2867± 0.3405	0.5205± 0.3677
5	0.9878± 0.7209	0.2076± 0.3864	0.3626± 0.3922	1.1616± 0.8854	0.3612± 0.5011	0.0000± 0.0000	0.6257± 0.7723	0.3105± 0.4208	0.5592± 0.6310	0.4745± 0.6487
6	1.0085± 0.5814	0.3181± 0.4593	0.1208± 0.3084	0.0318± 0.1388	0.3249± 0.3920	0.4731± 0.4727	0.0000± 0.0000	0.0911± 0.2173	0.2375± 0.3436	0.0565± 0.1697
7	0.0967± 0.2303	0.1884± 0.2880	0.5415± 0.5826	0.1646± 0.2856	0.8172± 0.7253	0.0954± 0.2275	0.0888± 0.3869	0.0000± 0.0000	0.2942± 0.3840	0.9142± 0.8677
8	0.8812± 0.7479	0.3788± 0.4533	0.5926± 0.6660	1.5277± 1.0709	0.5580± 0.6293	1.3725± 1.0884	0.1375± 0.4124	0.3683± 0.5750	0.0000± 0.0000	0.5580± 0.4780
9	0.1828± 0.2794	0.1241± 0.2488	0.1224± 0.2457	0.2153± 0.3439	1.3822± 1.2087	0.1846± 0.2825	0.1228± 0.2458	1.3587± 0.8656	0.1435± 0.3031	0.0000± 0.0000

Table 3: Confusion error rate in % where row is true label and column the predicted label.

5 hardest to predict correctly "pixelated images."

Here I print out the visualisations of the images that are most often incorrectly labelled(under the 20 runs with the optimal cross validated value of d) along with their true label.

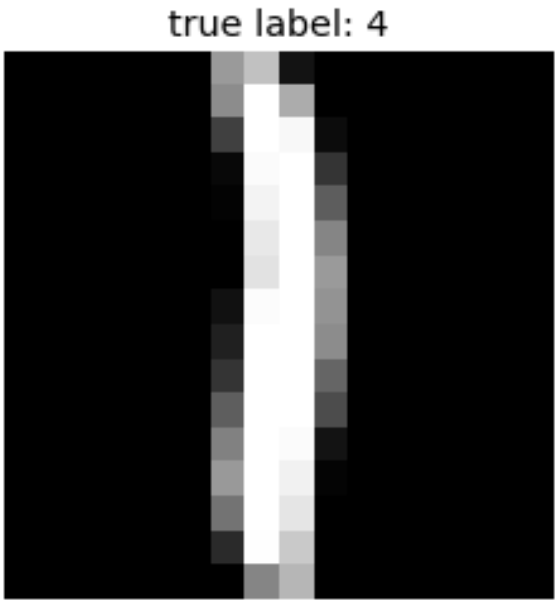

Figure 2: No surpise here, even a human would label this a 4.

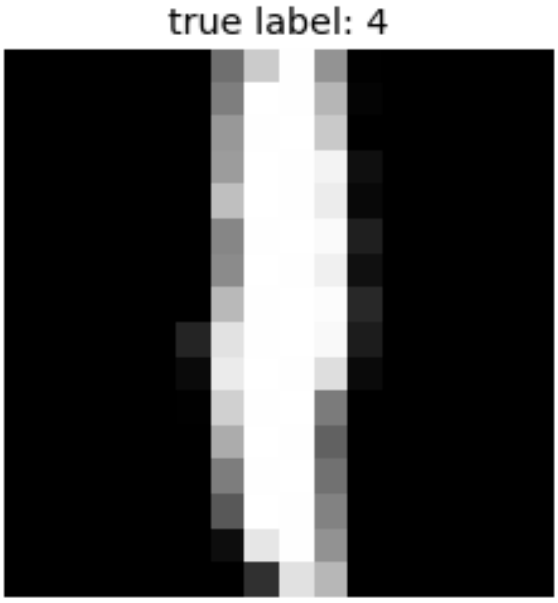

Figure 3: No surpise here, even a human would label this a 4.

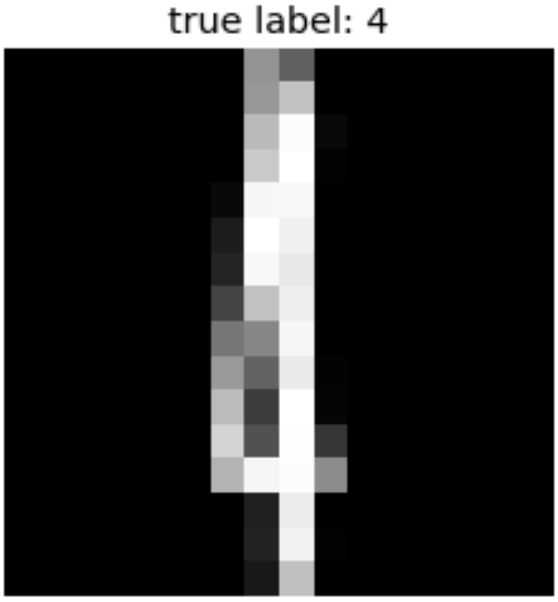

Figure 4: This one a human would label correctly however the 4 is very stretched so I'm not surprised the machine learner struggled. This could be seen as a 1 due to how it's elongated (and if you ignore the 'stem' of the 4)

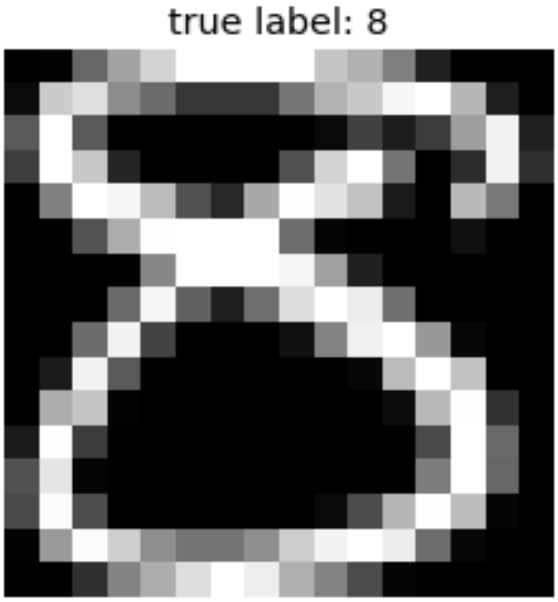

Figure 5: .This one is easy to label for a human. I suspect the little curl on the right top corner of the 8 confused the machine learner. I can see how with the curl, the 8 might be interpreted as a 5.

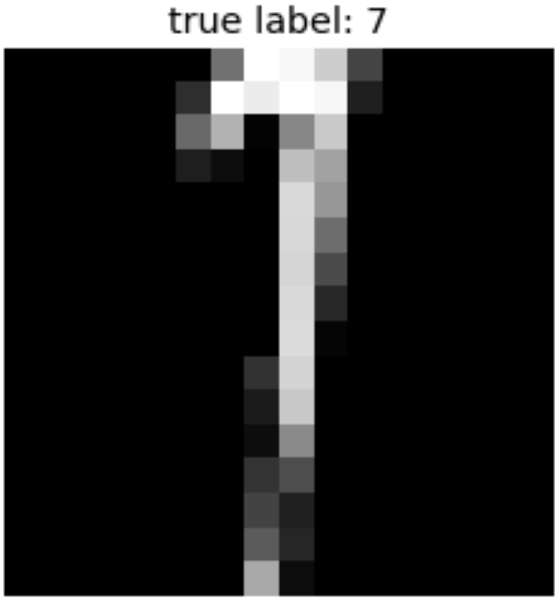

Figure 6: Not surprising that it was mislabelled. It can be easily confused with a 1 by a human.

Basic results using Gaussian Kernel $K(\mathbf{p}, \mathbf{q}) = e^{-c\|\mathbf{p}-\mathbf{q}\|^2}$

Here again I trained over 2 epochs. As an initial exploration to choose a reasonable set S of c-values (kernel width) to cross validate over, I conducted 5 runs with the data set randomly split into 80%-20% train-test. The exponential of -c*norm means that if we choose negative values of c, we risk the exponential exploding leading to overflow. Thus as an initial exploration I conducted the runs for c = -1,-0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2. The mean training accuracies for each of these is returned in Table 4.

c=-1.0	3.250
c=-0.75	3.186
c=-0.5	3.159
c=-0.25	3.111
c=0.0	10.507
c=0.25	99.642
c=0.5	99.653
c=0.75	99.629
c=1.0	99.624
c=1.25	99.624
c=1.5	99.624
c=1.75	99.626
c=2.0	99.621

Table 4:

As we can see negative c values are no good. And as soon as we exceed 0, the

accuracies sharply rise. However, the training accuracies start slowly decreasing after c=1.To avoid the risk of overfitting, I will not investigate beyond c=1 as the training accuracy at c=1 is already quite high. Instead I will further investigate values of c=0.5^n for n=0 to 15 to investigate what's happening between 0 and 1 for ever decreasing values of c. Again I will average the training error over 5 runs for 2 epochs. The training accuracies are returned in Table 5.

n=0	c=1.0	99.624
n=1	c=0.5	99.653
n=2	c=0.25	99.642
n=3	c=0.125	99.696
n=4	c=0.0625	99.769
n=5	c=0.03125	99.855
n=6	c=0.015625	99.796
n=7	c=0.0078125	99.336
n=8	c=0.00390625	97.871
n=9	c=0.001953125	95.454
n=10	c=0.0009765625	92.526
n=11	c=0.00048828125	89.622
n=12	c=0.000244140625	82.646
n=13	c=0.0001220703125	85.622
n=14	c=6.103515625e-05	67.552
n=15	c=3.0517578125e-05	59.524

Table 5:

As we can see we first pass the 90% accuracy threshold at n= 10 and from n= 10 to n= 5 the accuracies are mainly increasing with n=5 having the highest. Thus I will be cross validating values of c=0.5^n for n=5 to 10 and this will be S.But first we rerun some basic experiments on this new found S.

Here I performed 20 runs for d=1,...,7. Each run randomly splits the data into 80% train and 20% test. In Table 6 I report the mean test and train error rates as well as well as standard deviations. Here I trained for 2 epochs, as I'm training the OVR classifiers in parallel, each epoch consists of all 10 classifiers being trained on all training points. Thus to avoid possible overfitting and reduce training time, I chose 2 epochs so that each datapoint is seen and trained on at least twice.

n=5	n=6	n=7	n=8	n=9	n=10
c=0.03125	c=0.015625	c=0.0078125	c=0.00390625	c=0.001953125	c=0.000976563
0.1741± 0.0695	0.2373± 0.1356	0.6419± 0.2343	2.4802± 0.4829	4.5436± 0.9064	7.9661± 2.5635
3.3217± 0.4545	2.9398± 0.3665	3.5557± 0.4950	5.0538± 0.8802	6.4228± 1.0458	9.2200± 2.8364

Table 6: mean±std. Row 3 is training error, Row 4 is testing error.

As we can see, the bigger kernel widths in the table seem to have better training and testing errors. As n decreases, the testing error also goes down up to a point where

from n=6 to n=5 it actually decreases. Interesting to note that if we compare this to the polynomial kernel results from Table 1, we've managed to achieve smaller training and testing errors. Next we will try find the optimal gaussian kernel width.

Cross Validation

Next we perform 20 runs under the same conditions as above however this time, we take the training data and split it into 5-folds to do 5-fold cross validation and find the best parameter c^* over S. In Table 7 , we return the optimal value found in each of the 20 runs with their associated test errors.

run	optimal d	test error in %
1	0.03125	2.743
2	0.03125	3.281
3	0.015625	2.797
4	0.03125	3.55
5	0.03125	3.281
6	0.03125	3.066
7	0.03125	3.819
8	0.03125	3.712
9	0.03125	2.905
10	0.03125	3.604
11	0.03125	3.281
12	0.015625	2.259
13	0.03125	4.034
14	0.03125	2.743
15	0.03125	3.55
16	0.03125	2.69
17	0.03125	2.743
18	0.03125	4.088
19	0.03125	3.765
20	0.03125	3.497

Table 7:

Here is the 3.271±0.495(= mean test error±std) and 0.03±0.005(= mean d^* with std) for the above table.

The optimal test errors and std are slightly higher than in the polynomial kernel experiments, but nonetheless they are very close. I suspect that by cross validating further for a wider range of small c values(gaussian widths) might lead to a better set S that might reduce the errors.

1.3.2 OvO:One-vs.-one

Here is an alternate method to generalise the kernel perceptron to k-classes.

In One-vs.-One classification we train $\frac{(k)(k-1)}{2}$(=45 in our case) binary classifiers; one for each combination of two digit classes in our dataset. The binary classifier $h_{i,j}$(for i=0 to 9, for j=i to 9) will be trained to distinguish digits i and j on a reduced training dataset containing only instances of the two classes. Note that we if we trained a classifier $h_{i,j}$ we do not train another classifier $h_{j,i}$ as distinguishing digits i and j is the same as distinguishing digits j and i (symmetric). In this part again we will be using the polynomial kernel.

Algorithm 3 OVO multi-class kernel perceptron (training) for k-classes

Input $(\mathbf{x}_1, \mathrm{y}_1), \ldots, (\mathbf{x}_m, \mathrm{y}_m) \in R^n \times \{-1, 1\}$

Start a counter c to keep track of which classifier we're on
for $i = 0$ to k **do**
  for $j = i$ to k **do**
    if i=j **then** skip
    end if
    filter data so that we only keep instances of digits i and j
    compute Gram matrix K associated with the kernel function $K_{a,b} = K(x_a, x_b)$ for the filtered data.
    Initialise m_c by 1 array $\alpha = 0$ where m_c is the numberoffiltereddatapoints the cth classifier is trained on
    for $epoch = 1$ to $maxepochs$ **do**
      for $t = 1$ to $numberoffiltereddatapoints$ **do**
        y_t #receive label in OVO form
        Predict $\hat{y} = sign(matmul(\alpha, K[:, [t]]))$
        if prediction is wrong **then** update $\alpha =$
        end if
        Update α[t] = α[t] + y_t
      end for
    end for
    Store this current α in the cth position of the 45 by 1 cell array **A**
    increment the counter c that keeps track of which classifier we're on.
  end for
end for
Return **A**

Once we have the **A** containing 45 m_c by 1 arrays α corresponding to each of the 45 OVO binary classifier we can predict the label of a new point $x^* \in R^n$ by the following:

4. Compute the 45 (1 by m_c) kernel arrays $K_{1,a} = K(x_a, x^*)$ for a=1 to m_c (i.e. the size of the filtered set we trained on for each of the 45 classifiers).

5. Matrix multiply each of the 45 α arrays in **A** with the corresponding Kernel matrix that we computed above. This will give us 45 values that we will store as confidences

of each individual classifier in a vector κ. We trained the classifiers in a specific order, therefore these 45 confidences are also ordered. The 45 values are ordered such that they respectively correspond to training on digit pairs (0,1); (0,2); (0,3); (0,4); (0,5); (0,6); (0,7); (0,8); (0,9); (1,2); (1,3); ... (8,9). Thus when computing the confidences for a specific digit i, we sum all κ values that are associated to a classifier distinguishing (i, anything) and subtract all κ values that are associated to a classifier distinguishing (anything, i). The subtraction happens because (anything,i) is a classifier associated with distinguishing i however the labelling is reversed (+1 and -1 reversed) so the confidence values are also reversed in sign. Thus we subtract instead of summing these. Doing this process for each of the 10 digits in our dataset, gives rise to 10 digit confidences that we store in vector θ

6. Predict the label of x^* as $\text{argmax}_{i\in 1,2,,\ldots,k}\{\theta_i\}$.
The testing process is simple. We pass each of our testing points and using the above scheme we predict their labels. Then we just count how many of the labels we predicted incorrectly out of all the testing points. This then gives us our testing error. We can similarly find the training error by simply passing our training points to the prediction scheme instead.

Basic results using Polynomial Kernel $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q})^d$

Here I performed 20 runs for d=1,...,7. Each run randomly splits the data into 80% train and 20% test. In Table 8 I report the mean test and train error rates as well as well as standard deviations. Here I trained for 2 epochs as even though we train one smaller datasets, we do it 45 times (once for each classifier) and this makes the training time long.

d=1	d=2	d=3	d=4	d=5	d=6	d=7
14.64±1.62	4.56±0.66	2.33±0.25	1.39±0.16	1.02±0.14	0.81±0.13	0.67±0.12
15.28±1.94	6.58±0.79	4.8±0.59	4.13±0.48	3.82±0.46	3.67±0.46	3.49±0.45

Table 8: mean±std. Row 2 is training error, Row 3 is testing error.

As we can see, the increasing kernel power reduces the training error and testing error immensely. Both the training and testing errors monotonically decrease which suggests that we might not have encountered overfitting as opposed to the One Versus Rest scenario in Table 1. When compared to the One Versus Rest case, the corresponding training and test errors are higher however it might be worth looking into higher powers of d for One Versus One as the errors look like they might further decrease. Furthermore, it might also be worth running the training algorithm for more epochs which I decided not to do due to the long running time of the One Versus One implementation.

Cross Validation

Next we perform 20 runs under the same conditions as above however this time, we take the training data and split it into 5-folds to do 5-fold cross validation and find the best parameter d^*. In Table 9 , we return the optimal value found in each of the 20 runs with their associated test errors.

run	optimal d	test error in %
1	7	3.228
2	7	3.55
3	7	2.797
4	6	3.335
5	7	3.228
6	7	2.905
7	7	3.012
8	7	3.335
9	7	2.636
10	7	4.196
11	7	3.335
12	6	2.797
13	7	3.389
14	7	2.797
15	7	3.497
16	7	2.69
17	7	2.743
18	7	3.443
19	7	3.712
20	7	3.228

Table 9:

Here is the 3.193±0.388(= mean test error±std) and 6.9±0.3(= mean d^* with std) for the above table. This is actually lower than both the polynomial and gaussian kernel optimal test errors we had for One Versus Rest classification. It seems that the One versus one implementation is able to extract hidden data dependencies and better model the dataset without running into overfitting (as compared to OVR). The one versus one implementation however suffers from extreme computational costs so unless reducing the errors by as much as possible is required and we have a lot of time at hand for training, the OVR implementation seems more reasonable.

2 Part II: Semi-supervised Learning via Laplacian Interpolation

2.1 The data

The datasets used for this part are found in dtrain13_50.dat, dtrain13_100.dat, dtrain13_200.dat, dtrain13_400.dat. They consists of samples of the USPS digits dataset consisting of images from two classes (images of 0s and 2s). The difference between the 4 above datasets is just the size of them. In Figure 7, I display the dtrain13_50.dat dataset images as an example. Important to note is that all the datasets contain exactly equal amounts of images from each class.

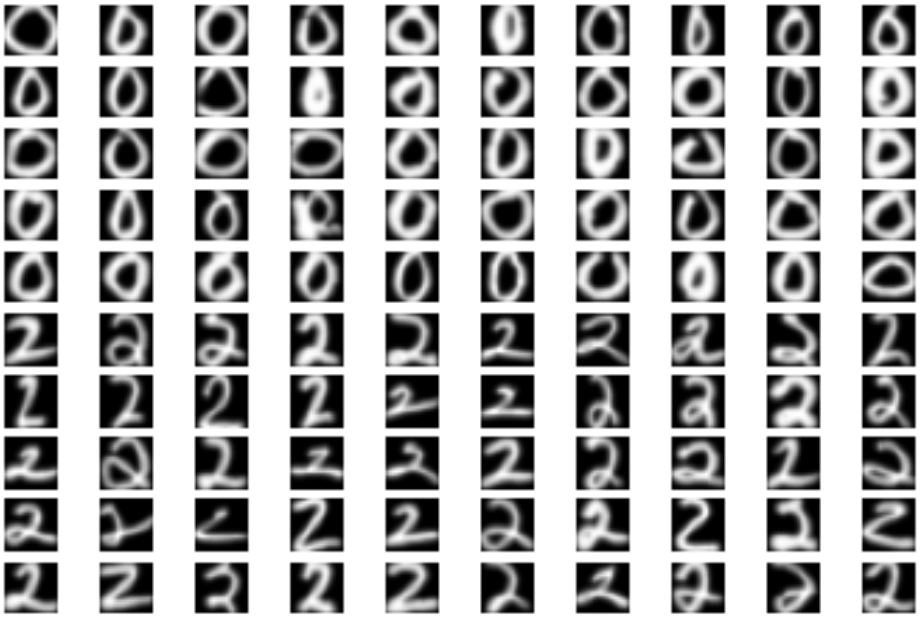

Figure 7: Images from the dtrain13_50.dat dataset.

2.2 Experimental report

Here I conducted 20 runs one each of the datasets. The Laplacian kernel interpolation was carried out using the scheme in the (Zhu et al. Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions) paper. In Table 10 I report the testing error and std for the Laplacian interpolation and in Table 11 I report the testing error and std for the Laplacian kernel interpolation.

	1	2	4	8	16
50	18.520±11.618	14.583±10.105	7.717±5.509	5.476±2.363	3.824±2.647
100	4.369±1.121	5.255±2.607	3.984±1.268	3.668±1.151	2.976±1.129
200	5.578±8.059	3.548±3.608	2.347±1.748	1.914±0.710	1.875±0.945
400	13.459±12.111	2.010±1.873	1.496±0.580	1.212±0.371	1.016±0.239

Table 10: Errors and standard deviations for semi-supervised learning via Laplacian interpolation. Row represents number of data points per label and column the number of known labels(for each class) in the training data.

	1	2	4	8	16
50	8.316±4.111	7.031±4.356	6.250±3.841	4.881±1.988	3.456±1.757
100	4.091±0.657	4.592±1.388	3.776±1.014	3.560±1.107	2.887±1.086
200	1.809±0.410	2.715±2.197	2.054±0.513	1.667±0.447	1.563±0.454
400	1.848±1.230	1.231±0.324	1.275±0.322	1.110±0.172	0.983±0.199

Table 11: Errors and standard deviations for semi-supervised learning via Laplacian(kernel) interpolation. Row represents number of data points per label and column the number of known labels(for each class) in the training data.

First thing to note in both Table 10 and Table 11 is that as we increase the number of known labels in the training set, the testing error significantly reduces. This is to be expected having more known labels boosts the semi-supervised learning capability. Second thing to note is that as we increase the the number of datapoints per label(i.e cycle through the 4 datasets), we also generally see a significant decrease in testing error(except in Table 1 for row 4 column 1). I suspect this is because as we increase the dataset size when we sample 2ℓ points from the dataset to form $\mathcal{L}$, the bigger size of the dataset makes it more likely that we sample representative terms from each class (samples from each class will be closer to the mean of each class) by the law of large numbers which makes us less likely to pick up noisy samples. This is especially true because the dataset have been sampled from the USPS dataset so they all have the same underlying distributions in terms of data. Next thing to note is that the mean and std test errors seem to always be lower in Table 11 when compared to respective counterparts in Table 10. This suggests that the laplacian kernel interpolation might be better performing than laplacian interpolation as we have lower test errors(laplacian kernel finds deeper relations in the data) and they seem to be more stable(not as volatile so less variance). The laplacian kernel seems to outperform the laplacian interpolation the most when we have a low number of known labels. For example when we have 4 or more known labels from each class, the values in the two tables are actually relatively close. I suspect that the kernel interpolation outshines the laplacian interpolation due to the way they learn to classify.

In the following, we focus on the Laplacian interpolation method, one of the established graph-based semi-supervised learning methods. The Laplacian interpolation method can be described as follows. We are given a data set which is represented by a data matrix $\mathbf{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)$ where each datum $\boldsymbol{x}_i \in \Re^n$ is a vector. Let $\mathbf{y} \in \{-1, 1\}^m$ be a label vector, where $y_i \in \{-1, 1\}$ is the label of $\boldsymbol{x}_i$. We assume that we know some of the values of y_i, and the task is to predict the rest. More formally, we are given $\{(i, y_i) : i \in \mathcal{L}\}$, where $\mathcal{L} \subset [m]$ is a set of indices of the known labels, and the task is to predict $\{(i, y_i) : i \in [m] \backslash \mathcal{L}\}$. In general, we only know the very few labels, i.e., $|\mathcal{L}| \ll m$. We will build a graph using the 3-NN method (with the Euclidean distance). For the 3-NN graph, we create a weight "1" edge if i-th datum is among the 3-nearest neighbors of j-th datum or if j-th datum is among the 3-nearest neighbors of i-th datum; otherwise, we put weight 0. Thus, we produce an $m \times m$ weight matrix as

$$W_{ij} := \begin{cases} \text{"+1"} & \boldsymbol{x}_i \text{ is 3-NN of } \boldsymbol{x}_j \text{ or } \boldsymbol{x}_j \text{ is 3-NN of } \boldsymbol{x}_i \\ \text{"0"} & \text{otherwise.} \end{cases}$$

Inputs:	Data: $\mathbf{X} := (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m), \boldsymbol{y} := (y_1, \ldots, y_m)$ Observed label set : $\mathcal{L} \subset [m]$
Adjacency weight matrix W**:**	$W_{ij} := \begin{cases} \text{“+1”} & \boldsymbol{x}_i \text{ is 3-NN of } \boldsymbol{x}_j \text{ or } \boldsymbol{x}_j \text{ is 3-NN of } \boldsymbol{x}_i \\ \text{“0”} & \text{otherwise} \end{cases}$ Note that $W_{ii} = 0$. ;
Degree Matrix D :	$D_{ii} := \sum_{j \neq i} W_{ij}$ and if $i \neq j$ then $D_{ij} := 0$.
Graph Laplacian L**:**	$L = D - W$,

Laplacian Kernel Interpolation (LKI):	Kernel Matrix $\boldsymbol{K} := (L^+_{ij} : i, j \in \mathcal{L})$ $\boldsymbol{y}_{\mathcal{L}} := (y_i : i \in \mathcal{L})$ $\boldsymbol{\alpha}^* := \boldsymbol{K}^+ \boldsymbol{y}_{\mathcal{L}}$ $\mathbf{v} := \sum_{i \in \mathcal{L}} \alpha^*_i \mathbf{e}_i^\top L^+$

As we can see, in LKI we compute the kernel matrix by restricting to the labelled set and even when computing $\mathbf{v}$ we'resumming over the labelled set. The way LKI builds the model is by investigating data dependencies between images in the labelled set and modelling the relationships between the labelled set images. Then it effectively takes this structural relationship and scales it up to include the unlabelled points to (i.e it almost assumes that the structural dependencies in the labelled set are a good approximation of the whole set's structure.
Whereas in Laplacian interpolation we do the following:

To compute the harmonic solution explicitly in terms of matrix operations, we split the weight matrix W (and similarly D, P) into 4 blocks after the lth row and column:

$$W = \begin{bmatrix} W_{ll} & W_{lu} \\ W_{ul} & W_{uu} \end{bmatrix} \tag{4}$$

Letting $f = \begin{bmatrix} f_l \\ f_u \end{bmatrix}$ where f_u denotes the values on the unlabeled data points, the harmonic solution $\Delta f = 0$ subject to $f|_L = f_l$ is given by

$$f_u = (D_{uu} - W_{uu})^{-1} W_{ul} f_l = (I - P_{uu})^{-1} P_{ul} f_l \tag{5}$$

Figure 8: From (Zhu et al. Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions)

So in Laplacian interpolation we do not look at the structural dependencies between labelled points among themselves at all. Instead we look at the structural depen-

dencies between unlabelled points mainly and the structural dependencies between unlabelled points and labelled points and then try to incorporate the two when building the model. Thus when we have a deficiency of labelled points, the Laplacian interpolation is more likely to struggle as compared to LKI as LKI is almost completely built from the labelled set (independent of the size of such), so the labelled set is used more extensively in the LKI. However, I suspect that this tendency of the LKI might be disadvantageous in certain scenarios. In this experiment, the the digits were 0 and 2 which are quite easy to distinguish(as compared to a 1 and 7 for example) as they have quite distinct shapes. Furthermore, the images were relatively low noise and the labelled set were sampled 1/2 from one class and 1/2 from the other (which is exactly the ratio they appear in the dataset as whole). Thus the structural model found by LKI on the labelled examples generalised quite well on the dataset as a whole. However, this might not be the case in more noisy training sets or training sets with class imbalances.

3 Part III: The 'just a little bit' problem

Problem ('just a little bit'): The m patterns $\boldsymbol{x}_1, \ldots \boldsymbol{x}_m$ are sampled *uniformly* at random from $\{-1, 1\}^n$, and each label is defined as $y_i := x_{i,1}$, i.e., the label of a pattern $\boldsymbol{x}$ is just its first coordinate. Thus for example here is a typical data set with $m = 4$ examples in $n = 3$ dimensions,

$$X = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix} \quad Y = \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

We are concerned with estimating the sample complexity as a function of the dimension (n) of the data of this problem. Where our "working definition" of sample complexity is the minimum number of examples (m) to incur no more than 10% generalisation error (on average).

(a) In this part I estimate the number of samples (m) to obtain 10% generalisation error versus dimension (n) for the perceptron, winnow, least squares, and 1-nearest neighbours algorithms. The results are averaged across 20 randomised runs for values of n up to 100 (except for 1-nearest neighbours where I stopped to n=14 due to the exploding running times in n).

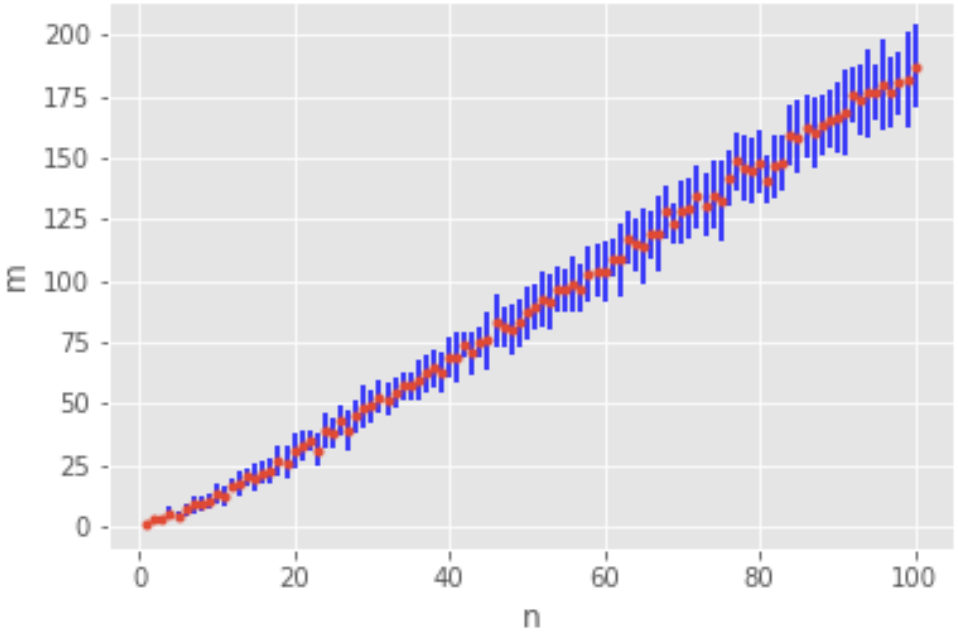

Figure 9: Estimated sample complexities(red) and standard deviations(blue) for perceptron.

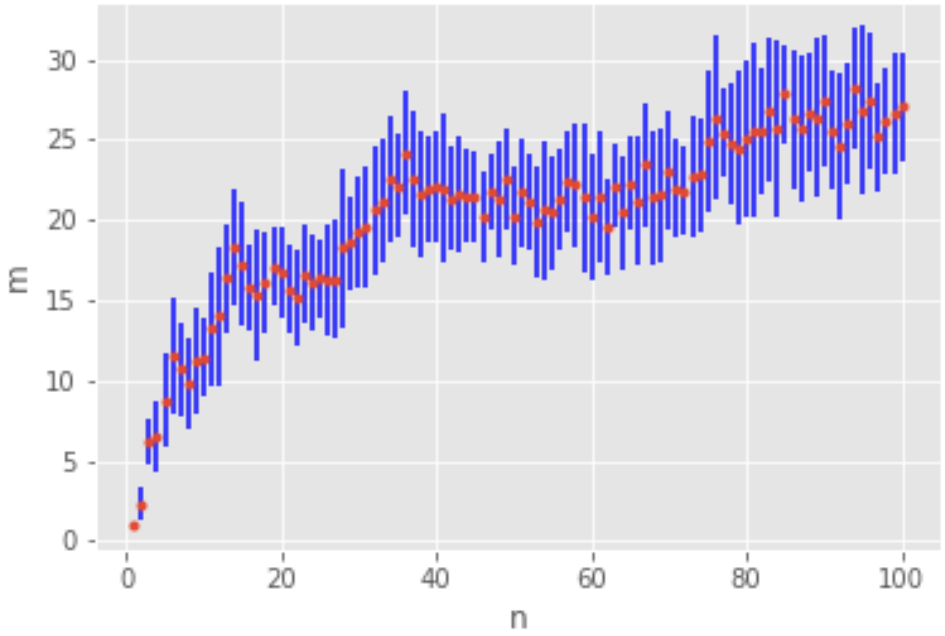

Figure 10: Estimated sample complexities(red) and standard deviations(blue) for winnow.

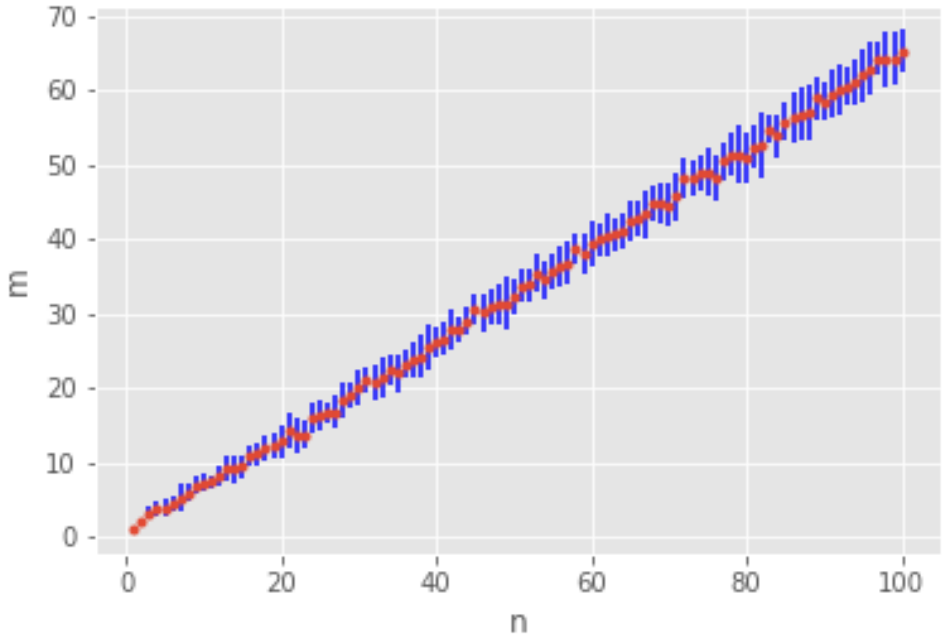

Figure 11: Estimated sample complexities(red) and standard deviations(blue) for least squares.

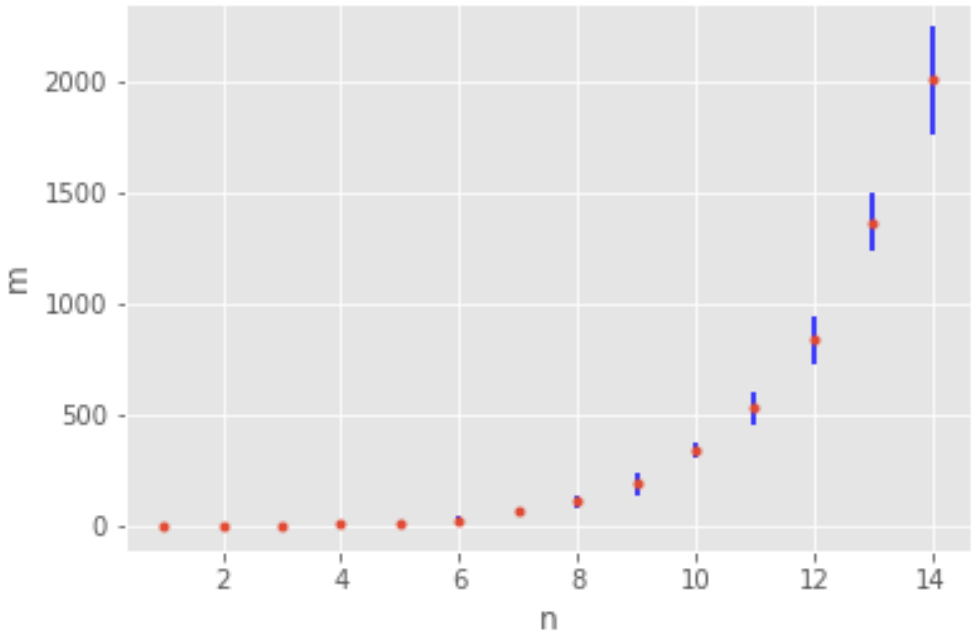

Figure 12: Estimated sample complexities(red) and standard deviations(blue) for 1-nearest neighbours.

(b) Computing the sample complexities exactly by simulation would be extremely expensive computationally. Thus I computed the empirical errors instead and

made use of the following from the lecture notes:

Model : Data is sampled IID from a distribution $\mathcal{D}$ (previously P) over $\mathcal{X} \times \mathcal{Y}$ with $\mathcal{Y} = \{0, 1\}$. The *expected error* (AKA *generalisation error*) of a function $h : \mathcal{X} \to \mathcal{Y}$ is

$$L_{\mathcal{D}}(h) = \mathcal{D}(\{(x, y) : h(x) \neq y\}) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[h(x) \neq y]$$

in our previous notation $L_{\mathcal{D}}(h) = \mathcal{E}(h) = \int [h(x) \neq y] dP(x, y)$.

The *empirical error* of h with respect to a data set $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ is denoted $L_S(h) = \sum_{i=1}^{m} \frac{1}{m}[h(x_i) \neq y_i]$ (previously $\mathcal{E}_{\text{emp}}(S, h)$).

Theorem: Select an h then for any $\delta \in (0,1)$. With probability at least 1-δ over the random sample V of size $|V|$ from D we have that $L_D(h) \leqslant L_V(h) + \sqrt{\frac{ln\frac{1}{\delta}}{2|V|}}$ Throughout the experiments my chosen value of δ was 0.01 so that the statement hold for probability at least 99%. For each given n, I increased m until the empirical error fell below $0.1 - \sqrt{\frac{ln\frac{1}{\delta}}{2|V|}}$ where I chose different $|V|$ for each of the 4 algorithms by a process of trial and error to reduce training times. Initially I was going to cross validate this value but it felt redundant as a process of trial and error would be good enough and the cross validation would have increased the running time. Thus, by choosing δ=0.01, I increased my confidence in the experiment results (accuracy) but this did lead to some longer computation times as now I needed the empirical error to fall below 0.1 - $\sqrt{\frac{ln\frac{1}{0.01}}{2|V|}}$ which is more restrictive.

(c) Next we try to estimate how m grows as a function of n as n goes to infinity for each of the algorithms.

Perceptron and Least Squares

Looking at Figure 9 and Figure 11, the sample complexities seems to be Θ(n).
From lectures:

Halfspaces: $\mathcal{X} = \mathbb{R}^n$, $\mathcal{H} = \{\mathbf{x} \mapsto \text{sign}(\langle \boldsymbol{w}, \mathbf{x} \rangle) : \boldsymbol{w} \in \mathbb{R}^n\}$

- Show that $\{\eta_1, \ldots, \eta_n\}$ is shattered
- Show that any $n + 1$ points cannot be shattered

Which is exactly describing the perceptron and least squares cases.

To show that $\mathrm{VCdim}(\mathcal{H}) = d$ we need to show that:

1. There exists a set C of size d which is shattered by $\mathcal{H}$.
2. Every set C of size $d+1$ is not shattered by $\mathcal{H}$.

Therefore both the perceptron and least squares have VCdim = n. Lastly:

Theorem (The Fundamental Theorem of Statistical Learning)

Let $\mathcal{H}$ be a hypothesis class of binary classifiers. Then, there are absolute constants C_1, C_2 such that the sample complexity of PAC learning $\mathcal{H}$ is

$$C_1 \frac{\mathrm{VCdim}(\mathcal{H}) + \log(1/\delta)}{\epsilon} \leq m_{\mathcal{H}}(\epsilon, \delta) \leq C_2 \frac{\mathrm{VCdim}(\mathcal{H}) \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

Furthermore, this sample complexity is achieved by the ERM learning rule.

Thus the $\frac{C_1}{0.1}(n+log(\frac{1}{\delta})) \leq m \leq \frac{C_2}{0.1}(nlog(\frac{1}{0.1})+log(\frac{1}{\delta}))$ for both the perceptron and least squares (with their corresponding C_1, C_2) which shows they're Θ(n).

Winnow

From lectures:

Theorem: Mistake bound of WINNOW (Littlestone)

The mistakes of WINNOW may be bounded by

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

if there exists a *consistent* k-literal monotone disjunction.

Which guarantees that Winnow is O(log(n)).

1NN

From lectures:

Corollary

Let $\mathcal{H}$ be a finite hypothesis class.

- *$\mathcal{H}$ is PAC learnable with sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leq \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$*
- *This sample complexity is obtained by using the $\mathrm{ERM}_{\mathcal{H}}$ learning rule*

In 1NN, each one of the 2^n points can be labelled in 2 ways thus we can have a total of 2^{2^n} possible classifiers. Therefore, $m \leqslant \frac{log(\frac{2^{2^n}}{\delta})}{0.1} \leqslant \frac{2^n log(2) - log(\delta)}{0.1}$. So 1NN is $O(2^n)$.

Theorem (Perceptron Bound [Novikoff])

For all sequences of examples

$$S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathbf{R}^n \times \{-1, 1\}$$

the mistakes of the PERCEPTRON algorithm is bounded by

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

with $R := \max_t \|\mathbf{x}_t\|$ when there exists a vector $\mathbf{v}$ with $\|\mathbf{v}\| = 1$ and constant γ such that $(\mathbf{v} \cdot \mathbf{x}_t) y_t \geq \gamma$ for all t.

(d) Since the label is completely determined by the first coordinate in our setup, we can just set $\mathbf{v}$= (1,0,...,0) with a 1 only in the first position and set $\gamma = 1$ so that $(\mathbf{v} \cdot \mathbf{x}_t) y_t \geqslant \gamma$ for all t. Furtermore, R = $\max_t \|x_t\| = \sqrt{n}$ as $x_t \in \{-1, 1\}^n$. Therefore the mistakes of the perceptron is bounded by $M \leq \frac{n}{\gamma^2} \leq n = B$.

Theorem

Given m, then let t be drawn uniformly at random from $\{1, \ldots, m\}$. Let $\mathcal{S}$ consist of t examples sampled iid from P, let $(\mathbf{x}', y')$ be an additional example sampled from P then

$$\mathrm{Prob}(\mathcal{A}_{\mathcal{S}}(\mathbf{x}') \neq y') \leq \frac{B}{m}$$

with respect to the draw of t, $\mathcal{S}$, and $(\mathbf{x}', y')$.

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