Supervised Learning - Assignment 2

21180859 and 21059917
Group: Yes

Contents

1 Part I

In this part we analyze the multi-class kernel perceptron equipped with the polynomial and Gaussian kernel. We will now discuss these two kernels and how they are implemented as well as the two methods used for generalizing the binary classification problem to a k-class classification problem.

1.1 A discussion of our implementation of the kernel perceptron

The polynomial kernel is a kernel function $K : x \times y \to R$ such that $K(x, y) = (x^T y + r)^d$ where $x, y \in R^n$ and $r \in R$. For $c > 0$, the Gaussian Kernel is defined as: $K(x, y) = e^{-c\|x-y\|^2}$ where c denotes the width of the Gaussian Kernel. The parameter c controls the flexibility of kernel similar to d in the Polynomial kernel. That is, a large c is equivalent to a large d, which allows the classifiers to fit any labels, whereas a small c may reduce the Gaussian kernel to a constant, which may result in a failure to learn any non-trivial classifier [3].
When applying a kernel to all of the datapoints in a data matrix $X \in R^{m\times n}$, one can create a compact representation, called the kernel matrix:

$$K(\mathbf{X}, \mathbf{X}) = \begin{pmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & K(\mathbf{x}_1, \mathbf{x}_2) & \cdots & K(\mathbf{x}_m, \mathbf{x}_m) \\ K(\mathbf{x}_2, \mathbf{x}_1) & K(\mathbf{x}_2, \mathbf{x}_2) & \cdots & K(\mathbf{x}_2, \mathbf{x}_m) \\ \vdots & \vdots & \ddots & \vdots \\ K(\mathbf{x}_m, \mathbf{x}_1) & K(\mathbf{x}_m, \mathbf{x}_2) & \cdots & K(\mathbf{x}_m, \mathbf{x}_m) \end{pmatrix}$$

This matrix representation will help us explain the sum $w(\dot{}) = \sum_i^m \alpha_i K(x_i, \dot{})$ later.

1.1.1 Prediction

The perceptron is an online algorithm which takes in data $(X, y) \in R^{m\times n} \times R^{m\times 1}$ and updates its parameters using information from 1 data point at a time. The model learned by standard perceptron algorithms is a linear binary classifier and in order to generate a nonlinear separating surface we use a kernel perceptron.
For a two class classifier, we predict with:

$$\hat{y}_t = \text{sign}(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)) = \text{sign}\left((\alpha_1, \alpha_2, \cdots, \alpha_{t-1}) \begin{pmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_{t-1}, \mathbf{x}_t) \end{pmatrix} \right)$$

We use two methods for generalising the binary classifiers to multi-class classifiers: the one-vs-rest and the one-vs-one method [1].

One versus Rest:

The one-vs-rest approach [2] involves using one classifier per class, where the examples of that class are used as positive examples and others as negative examples. Each of the classifiers then produces a confidence score and the prediction is equal to the class of which classifier produces the highest confidence score. Hence, when there are k classes we use k classifiers of dimension m, where m is the number of examples in the training data. The classifiers can be represented as a matrix, where each row represents the weights of the classifier:

$$\alpha = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \cdots & \alpha_{1,m} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \cdots & \alpha_{2,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{k,1} & \alpha_{k,2} & \alpha_{k,3} & \cdots & \alpha_{k,m} \end{pmatrix}$$

Hence, to classify a new data point $\mathbf{x}_t$ we evaluate the following expression:

$$\text{argmax}_k \left(\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & \cdots & \alpha_{1,m} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \cdots & \alpha_{2,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{k,1} & \alpha_{k,2} & \alpha_{k,3} & \cdots & \alpha_{k,m} \end{pmatrix} \begin{pmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_m, \mathbf{x}_t) \end{pmatrix} \right) = \text{argmax}_k \begin{pmatrix} \alpha_{1,1}K(\mathbf{x}_1, \mathbf{x}_t) + ... + \alpha_{1,m}K(\mathbf{x}_m, \mathbf{x}_t) \\ \alpha_{2,1}K(\mathbf{x}_1, \mathbf{x}_t) + ... + \alpha_{2,m}K(\mathbf{x}_m, \mathbf{x}_t) \\ \vdots \\ \alpha_{k,1}K(\mathbf{x}_1, \mathbf{x}_t) + ... + \alpha_{k,m}K(\mathbf{x}_m, \mathbf{x}_t) \end{pmatrix}$$

One versus One:

The one-vs-one approach involves training $K(K-1)/2$ binary classifiers of dimension m. Each classifier learns to distinguish between a pair of classes. For every data point, each classifier casts a vote for the class they predict and the predicted label is then the most voted for class. The matrix which represents the vector representation of the classifiers takes size $\frac{k\times(k-1)}{2} \times m$. The classifiers in the one-vs-one method try to distinguish between the following pairs:
$[0,1], [0,2], [0,3], ..., [0,9], [1,2], [1,3], ..., [1,9], ..., [8,9]$. These pairs represent all the binary combinations of the k classes. As mentioned previously, the standard (binary) perceptron classifies the points of its inputs to two classes, the positve and the negative classes. Since in our one-vs-one classification algorithm we use $\frac{k(k-1)}{2}$ of these standard perceptrons, we assume the first class of each pair represents the positive class and the second class represents the negative class. To classify a new data point x_t, we first evaluate the following expression:

$$\text{sign}\left(\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,m} \\ \alpha_{2,1} & \alpha_{2,2} & \cdots & \alpha_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ \alpha_{\frac{k(k-1)}{2},1} & \alpha_{\frac{k(k-1)}{2},2} & \cdots & \alpha_{\frac{k(k-1)}{2},m} \end{pmatrix} \begin{pmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_m, \mathbf{x}_t) \end{pmatrix}\right) = \begin{pmatrix} \text{sign}(\sum_{i=1}^{t-1} \alpha_{1,i} K(\mathbf{x}_i, \mathbf{x}_t)) \\ \text{sign}(\sum_{i=1}^{t-1} \alpha_{2,i} K(\mathbf{x}_i, \mathbf{x}_t)) \\ \vdots \\ \text{sign}(\sum_{i=1}^{t-1} \alpha_{\frac{k(k-1)}{2},i} K(\mathbf{x}_i, \mathbf{x}_t)) \end{pmatrix}$$

and then count the number of times each class has been voted for. The class with the most votes is the predicted class.

1.1.2 Update Step

Binary Classifier:

1. if $\hat{y}_t = y_t$ then α_t=0, else $\alpha_t = y_t$,
2. $w_{t+1}(\cdot) = w_t(\cdot) + \alpha_t K(x_t, \cdot)$

One versus Rest:

1. If $\hat{y}_t \neq y_t$, update alpha matrix the following way:
2. $\alpha_{\hat{y}_t,t} = \alpha_{\hat{y}_t,t} - 1$ (decreasing the confidence of the wrong predicted classifier).
3. $\alpha_{y_t,t} = \alpha_{y_t,t} + 1$ (increasing the confidence of the real classifier).

One versus One:

1. If $\hat{y}_t \neq y_t$, then we update all classifiers α_i which have y_t (the real label) in their pair the following way:
2. $\alpha_{i,t}+ = 1$ if y_t is the positive class with respect to this classifier
3. $\alpha_{i,t}- = 1$ if y_t is the negative class with respect to this classifier

1.1.3 Limitations of the multi-class methods

There are various problems with both of these methods. The first problem being of which both methods suffer is that they produce ambiguous regions [1]. A problem with the one-vs-rest classifier is that each classifier sees unbalanced data because the set of negative values they see is much larges than the positive ones. Lastly, a problem with the one-vs-one classifier is that the more classes there are the more significant the amount of time it takes to compute.

1.1.4 Selection of the number of epochs

The only parameter which we did not cross validate over was the number of epochs. We determined that after the 10th epoch the error decrease plateaued in every model we trained. In table 1 you will be able to see the error for each epoch per dimension d. The horizontal axis represents the polynomial dimension and the vertical axis the epoch. Note that we provide a table and not a plot so that's it's clear that the incremental error decrease is small.

Table 1: Training error in each epoch versus polynomial dimension d

	1	2	3	4	5	6	7
1	15.20	10.67	9.11	8.41	8.06	7.96	7.92
2	9.86	4.43	2.74	2.17	1.82	1.63	1.53
3	8.82	2.80	1.51	1.00	0.77	0.61	0.55
4	8.18	2.06	0.97	0.60	0.40	0.32	0.29
5	7.78	1.57	0.67	0.42	0.28	0.19	0.19
6	7.42	1.25	0.49	0.27	0.20	0.16	0.12
7	6.98	1.03	0.41	0.22	0.13	0.11	0.10
8	6.88	0.86	0.30	0.17	0.13	0.08	0.09
9	6.60	0.66	0.28	0.13	0.11	0.08	0.06
10	6.45	0.65	0.24	0.14	0.08	0.07	0.06

It is clear that the error decrease plateaus and even increases for $d = 4$ after 9th epoch, hence we decided to stop at 10 epochs.

Furthermore, we experimented with the method mentioned in [4]. To explain their method, we first introduce some notation: $E_{va}(t)$ is the validation error at time t, $E_{opt}(t)$ is defined to be the lowest validation set error obtained in epochs up to t i.e., $E_{opt}(t) := \min_{t'<t} E_{va}(t')$. We now define the generalization loss at epoch t to be the relative increase of the validation error over the minimum-so-far (in percent):

$$GL(t) = 100 \left(\frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

We stop training as soon as the generalization loss exceeds a certain threshold α and pick the previous epoch's classifiers as the optimal classifiers.

This method turned out to be inferior to the simple method of using just 10 epochs, as the trade-off between results and computation time was not worth it.

1.2 Experiments

In this section we describe the experiments we did in order to analyze the kernel perceptron.

1.2.1 Basic Results

One versus Rest: Polynomial and Gaussian Kernels

We now present a table which corresponds to the basic results using the one-vs-rest method with both kernels:

Table 2: Comparison between Gaussian and Polynomial Kernel experiment 1

Polynomial			Gaussian		
d	Train error %	Test error %	c	Train error %	Test error %
1	6.0480 ± 1.4855	6.6676 ± 1.4855	0.01	0.0874 ± 0.1162	0.6563 ± 0.1162
2	0.5902 ± 0.3567	1.3475 ± 0.3567	0.02	0.0229 ± 0.0245	0.5783 ± 0.0245
3	0.1250 ± 0.0763	0.8203 ± 0.0763	0.03	0.0134 ± 0.0095	0.6509 ± 0.0095
4	0.0538 ± 0.0383	0.6428 ± 0.0383	0.04	0.0235± 0.0394	0.6859 ± 0.0394
5	0.0303 ± 0.0265	0.5621 ± 0.0265	0.05	0.0208 ± 0.0547	0.7800 ± 0.0547
6	0.0679 ± 0.1428	0.7558 ± 0.1428	0.06	0.0168 ± 0.0526	0.9118 ± 0.0526
7	0.0249 ± 0.0214	0.6643 ± 0.0214	0.07	0.0047 ± 0.0149	0.8553 ± 0.0149
			0.08	0.0007 ± 0.0029	0.9118 ± 0.0029
			0.09	0.0000 ± 0.0000	0.9441 ± 0.0000
			0.1	0.0000 ± 0.0000	1.0005 ± 0.0000

For the polynomial kernel we can see that as the dimension of the polynomial increases the error also decreases. This is due to the fact that at higher dimensions the mapped data is more linearly separable than the lower ones. For the Gaussian kernel before doing question 1 we have searched over the interval $\{(0.1)^k\}$ for $k \in \{1, ..., 10\}$ and found that we get the best errors in the interval $[0.01, 0.1]$ and decided to analyze the error in the set $\{0.01, 0.02, ..., 0.1\}$. From table 2 it is evident that as c increases the training error decreases as well. For the last value of c we observe non-existent training error which is lower than the lower values of c, however due to overfitting the testing error is 30 to 40 % higher than the previous value of c's, since the parameters were overfit and didn't generalize as well. Since the Gaussian kernel has an infinite feature map dimension, it does a better job at separating the data than the polynomial kernel, which has a finite feature map dimension.

One versus One

In table 3 we present the basic results of the one-vs-one method equipped with the polynomial kernel:

Table 3: One versus One method basic results

d	Train error %	Test error %
1	2.9483 ± 0.7140	3.7305 ± 0.7140
2	0.1250 ± 0.0804	0.7880 ± 0.0804
3	0.0376 ± 0.0274	0.7046 ± 0.0274
4	0.0201 ± 0.0131	0.7315 ± 0.0131
5	0.0363 ± 0.0306	0.6508 ± 0.0306
6	0.0188 ± 0.0161	0.6966 ± 0.0161
7	0.0262 ± 0.0286	0.7315 ± 0.0286

One versus One (OVO) compared to One versus Rest (OVR) is shown below:

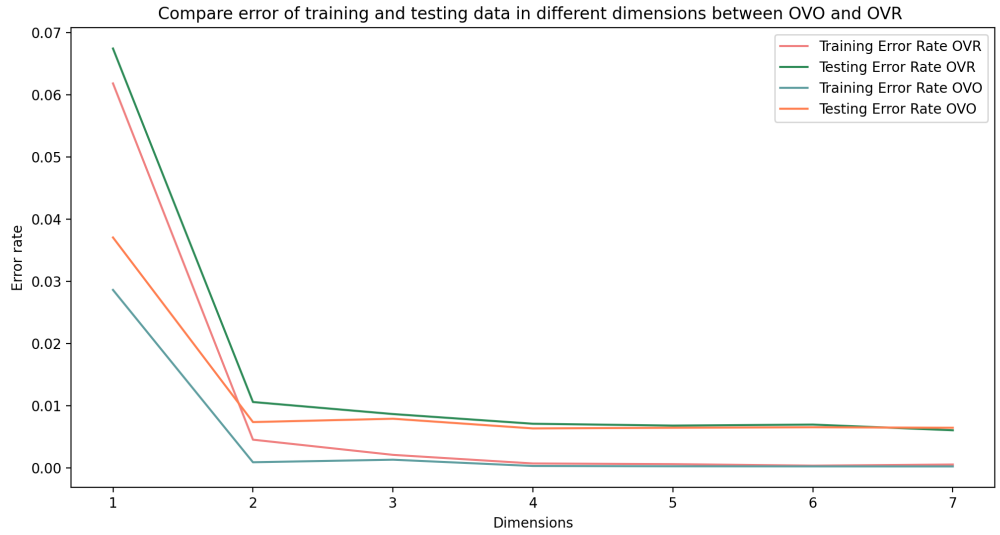

Fig 1

We can see in Fig 1 that the training error (pure values, not percentages) of OVO is slightly higher than OVR, while the test error of OVO is lower than that of OVR. That is, in this experiment OVO performs better than OVR. This is quite reasonable because OVO performs better when handling imbalanced data [5]. In the given dataset, there are more examples with labels 0 and 1 (approximately 500 more) than the other examples, resulting in a skewed distribution.

1.2.2 Cross-Validation

In order to find well-performing parameters in terms of generalization error, we perform 20 runs of 5-fold cross-validation on training data. In each run we split the training data into 5 folds and obtain validation error for every value the parameters can take. We then pick the best parameters for each run. In table 4 you will see the average results for the 20 runs:

Table 4: Cross-Validation Results

Method	Test error %	Parameter
OvR Polynomial	0.7073 $\pm$ 0.2057	d = 4.85 $\pm$ 1.0136
OvR Gaussian	0.5675 $\pm$ 0.144815	c = 0.015 $\pm$ 0.005
OvO Polynomial	0.6589 $\pm$ 0.2223	d = 3.9 $\pm$ 1.1136

From table 4, we can observe that the mean test error for OVO is smaller than that of OVR. Additionally, the best polynomial degree of OVR is 4.85, which is higher than the of OVO, which is 3.9. This implies that OVO requires a less complex hypothesis than OVR.

1.2.3 Confusion Matrix

In the Figure 2 you will see the rate at which numbers are being wrongly predicted and which numbers they are "confused" with (not that the scale are pure values if the error rate, not percentages):

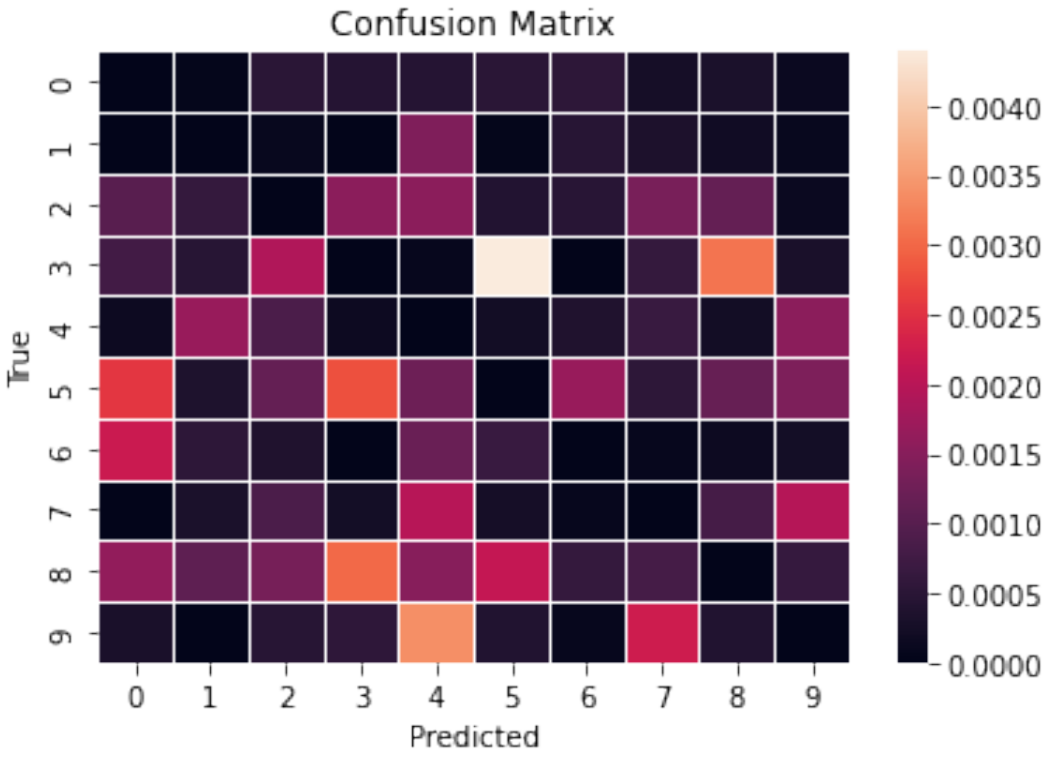

Fig 2

In table 5 you will see the exact numerical values of the rate of confusion and their respective standard deviations:

It is obvious that the numbers which look alike get confused more often with each other. There are some digits that mis-classified more frequently than other. For instance, 3 has higher probability to be classified as 5 and 8 has higher probability to be classified as 3.

Table 5: Confusion Matrix

	0	1	2	3	4
0	0.0+-0.0	0.02+-0.0007	0.06+-0.0013	0.02+-0.0007	0.02+-0.0007
1	0.0+-0.0	0.0+-0.0	0.02+-0.0009	0.0+-0.0	0.1+-0.0022
2	0.08+-0.002	0.06+-0.0018	0.0+-0.0	0.08+-0.0019	0.17+-0.0025
3	0.06+-0.0017	0.0+-0.0	0.09+-0.0027	0.0+-0.0	0.0+-0.0
4	0.0+-0.0	0.29+-0.0043	0.12+-0.003	0.0+-0.0	0.0+-0.0
5	0.31+-0.006	0.0+-0.0	0.04+-0.0015	0.23+-0.0038	0.14+-0.0028
6	0.12+-0.0023	0.16+-0.0033	0.03+-0.0013	0.0+-0.0	0.07+-0.002
7	0.0+-0.0	0.06+-0.002	0.09+-0.0022	0.04+-0.0016	0.26+-0.0037
8	0.17+-0.003	0.17+-0.003	0.2+-0.0036	0.32+-0.0043	0.07+-0.002
9	0.03+-0.0013	0.06+-0.0017	0.06+-0.0019	0.0+-0.0	0.15+-0.0026
	5	6	7	8	9
0	0.03+-0.001	0.05+-0.0012	0.02+-0.0007	0.0+-0.0	0.0+-0.0
1	0.0+-0.0	0.02+-0.0009	0.04+-0.0013	0.0+-0.0	0.02+-0.0009
2	0.03+-0.0012	0.05+-0.0016	0.22+-0.0031	0.08+-0.0019	0.03+-0.11
3	0.66+-0.0059	0.0+-0.0	0.03+-0.0013	0.18+-0.0027	0.0+-0.0
4	0.03+-0.0012	0.15+-0.0032	0.09+-0.0021	0.0+-0.0	0.29+-0.0038
5	0.0+-0.0	0.21+-0.0031	0.07+-0.0021	0.1+-0.0025	0.07+-0.0021
6	0.0+-0.0	0.0+-0.0	0.0+-0.0	0.03+-0.0013	0.0+-0.0
7	0.06+-0.0018	0.0+-0.0	0.0+-0.0	0.03+-0.0014	0.32+-0.0037
8	0.25+-0.004	0.03+-0.0015	0.04+-0.0016	0.0+-0.0	0.1+-0.0024
9	0.0+-0.0	0.0+-0.0	0.3+-0.0053	0.06+-0.0018	0.0+-0.0

1.2.4 Five Hardest to Predict Correctly "Pixelated Images"

In Figure 3 you will see the numbers which are hard to predict i.e., our algorithms have the most trouble classifying these:

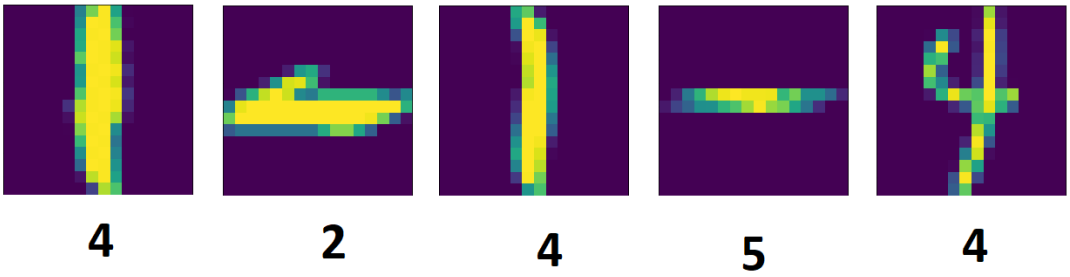

Fig 3 *Hardest to predict numbers*

It is obvious that these numbers are poorly drawn and not even a human could classify them correctly.

2 Part II

Spectral Clustering

Spectral clustering emerged from the theory of spectral graph theory. Its basic idea consists of developing a weighted graph from the original data set where data points are treated as nodes in the graph [6]. In the context of spectral clustering, clustering can be viewed as a graph cut problem. As a result, it can separate irregular shaped groups and cluster irregular data clusters.

Algorithm Implementation

Let $\mathbf{X} = \mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_l$ be a set of patterns to be clustered, where $\mathbf{x}_i \in R^n$. Then we can build a complete weighted undirected graph G(V,E) where $V := \{v_1, v_2, \cdots, v_l\}$ represents data points and E represents a set of edges [7]. We assume G as weights that measure each edge between two vertices v_i and v_j. The weighted adjacency matrix of the graph denoted as $\mathbf{W}_{l\times l}$ is:

$$W = (e^{-c\|\mathbf{x}_i - \mathbf{x}_j\|^2})_{i,j=1}^{l}$$

where $\mathbf{x}_i$ and $\mathbf{x}_j$ are roughly in the same cluster when $w_{i,j}$ is large. Then we construct the graph Laplacian $L = D - W$, where D is a diagonal matrix where $D(i,i) = \sum_{j=1}^{l} W(i,j)$ with the non-diagonal elements equal to 0. We will use the eigenvector ($\mathbf{v}_2$) corresponding to the second smallest eigenvalue of L to perform clustering in the following way: Assign $\mathbf{x}_i$ to cluster '+1' when $\text{sign}(\mathbf{v}_2(i)) \in \{0, 1\}$ and '-1' otherwise.

2.1 Implementing the spectral clustering on twomoons

In this part we perform spectral clustering on twomoons data with the kernel parameter c taking values in the following set:

$$c \in \{2^i : i \in \{-10.0, -9.9, -9.8, \cdots, 9.9, 10\}\}$$

Original data shows in Figure 4:

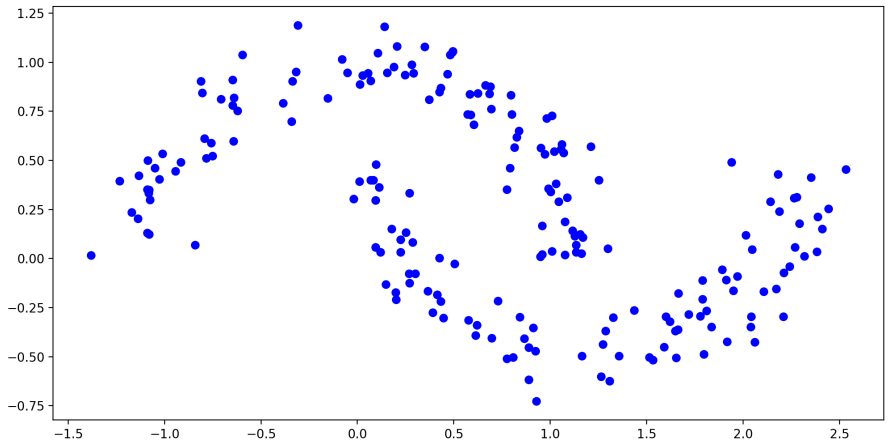

Fig 4 *Twomoons Original Data*

After performing spectral clustering with different values of c, we chose the one which correctly clustered the data and plotted it in Figure 5:

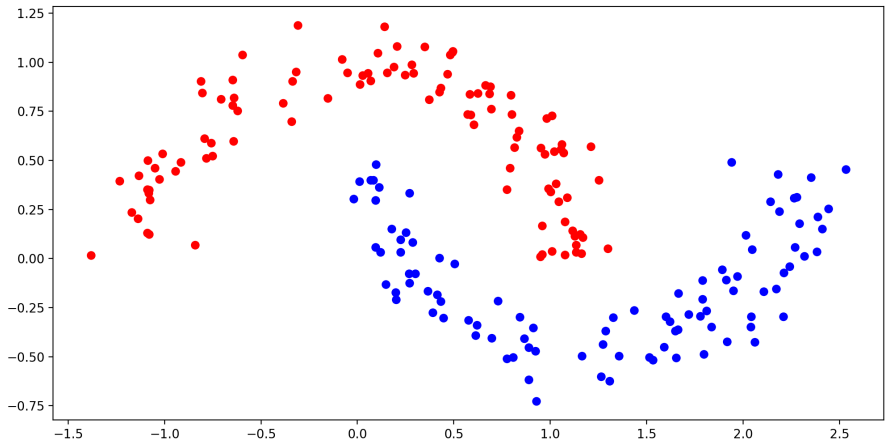

Fig 5 *Twomoons Clustered*

where $c \approx 21$

2.2 Implement spectral clustering on isotropic Gaussian data

Consider the traditional gaussian distribution:

$$N(\mu, \Sigma)$$

An isotropic gaussian is one where the covariance matrix is represented by the simplified matrix $\Sigma = \sigma^2 I$, which means all dimensions are independent and the variance of each dimension is the same. As a result, a set of isotropic gaussian data should become circular centered at mean location $u = (x_0, y_0)$. The plot of original data and clustered data is shown in Fig 6 below:

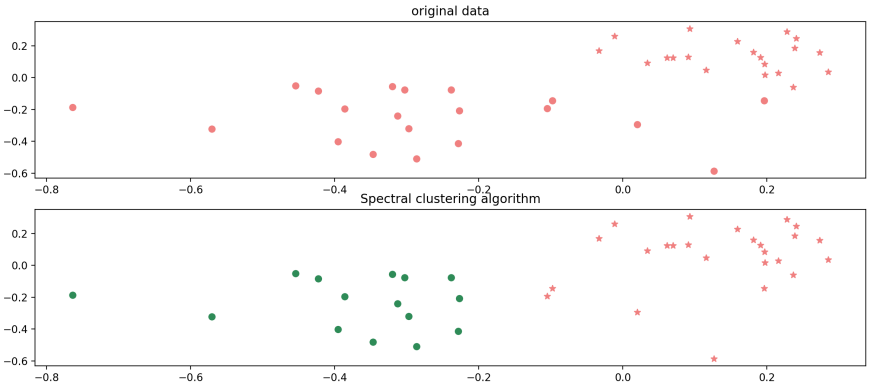

Fig 5 *Isotropic gaussian original and clustered data*

In this subsection we chose to implement the algorithm with vaule of c ranging in the following interval:

$$c \in \{0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 1.5, 2\}$$

and $c = 0.5$ gave almost correctly cluster data which made 5 mistakes.

2.3 Test the spectral clustering on real digit data

In this subsection we cluster digit data into class -1 and 1, where '3' represents -1 and '1' represent 1 with $classy_i = 2 \times I(y_i = 1) - 1$. In order to choose best c we use the correct cluster percentage:

$$CP(c) = \frac{max\{l_-, l_+\}}{l}$$

where l_- denotes the number of wrong predictions and l_+ denotes the number of correct predictions. Here we choose c from 0 to 0.1 with step size 0.002 and produce a plot which shows the correctness of c versus its values in the above specified set:

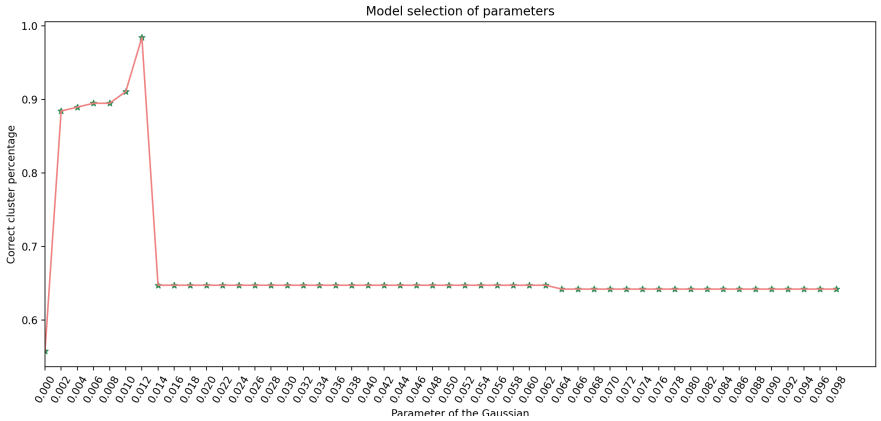

Fig 6 *Correctness versus c*

From Figure 6 we can observe that the best c is around 0.012 and the correct cluster percentage will remain the same after $c = 0.014$.

2.4 Explain why CP(c) is a reasonable measure

We think the best approach to explaining this would be through an example. If we go back to figure 5, suppose WLOG that that the original labels of the red and blue points are class 1 and 3 respectively. Now, suppose our algorithm classifies all of the red points as blue (as class 3) and all of the blue points as red (as class 1). If we were trying to classify points (in a classification problem), this result would be a total failure with error equal to 100 %. However, since our goal is to place

homogeneous points in the same cluster, this result (in terms of a clustering problem) yielded a perfect score, since every point was placed in a cluster where there exists only 1 class, meaning our algorithm correctly (100 % accurately) spotted the presence of 2 clusters out of 2.

Now, CP(c) tells us exactly the same as what we just described above (at least in a scenario with 2 classsess/clusters). l_- measures the amount of labels which were incorrectly classified i.e., the original label is not the same as its respective predicted label and l_+ measures the amount of labels correctly classified. Assume we had 100 points of each class in the above example. Then l_- would be equal to 200 and l_+ would be 0. Then the CP(c) measure would indicate that the correctly clustered percentage is 100 % and this makes sense since our ultimate goal was to place all of the same labels in one cluster. In the opposite case where the algorithm classified all of the red points as red and blue as blue, we would again get a CP(c) measure of 100 % with $l_+ = 200$ and $l_- = 0$

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

From the definition of eigenvector and eigenvalue, we have:

$$L\mathbf{v} = \lambda\mathbf{v}$$

where L denotes the laplacian matrix and $\lambda, \mathbf{v}$ represent L's eigenvalue and eigenvector respectively.

Let $\mathbf{1}_{l\times 1} := \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$, and since $L = D - W$, we have that:

$$L\mathbf{1} = (D - W)\mathbf{1}$$

Since

$$D = \begin{pmatrix} d_{11} & 0 & \cdots & 0 \\ 0 & d_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{ll} \end{pmatrix}$$

where $d_{ii} = \sum_{i=1}^{l} W_{i,j}$. Hence, we can compute $D\mathbf{1}$ as:

$$D\mathbf{1} = \begin{pmatrix} d_{11} & 0 & \cdots & 0 \\ 0 & d_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{ll} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} d_{11} \\ d_{22} \\ \vdots \\ d_{ll} \end{pmatrix}$$

We can also obtain $W\mathbf{1}$:

$$W\mathbf{1} = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1l} \\ w_{21} & w_{22} & \cdots & w_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ w_{l1} & w_{l2} & \cdots & w_{ll} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} w_{11} + \cdots + w_{1l} \\ w_{21} + \cdots + w_{2l} \\ \vdots \\ w_{l1} + \cdots + w_{ll} \end{pmatrix} = \begin{pmatrix} d_{11} \\ d_{22} \\ \vdots \\ d_{ll} \end{pmatrix} = D\mathbf{1}$$

Therefore $L\mathbf{1} = (D - W)\mathbf{1} = 0\mathbf{1}$, where 0 is one of the eigenvalues and $\mathbf{1}$ is one of the eigenvectors of L. Since L is a positive semi-definite matrix, all of the eigenvalues of L are larger or equal than 0. As a result, 0 is smallest eigenvalue and the corresponding eigenvector is the constant vector with all elements equal to 1.

2.6 Explain why spectral clustering 'works'

The spectral clustering works because of the spectral decomposition of the Laplacian matrix. Laplacian can be viewed as an operator on functions defined on G because of symmetric, positive semi-definite and the decomposition of Laplacian matrix can show the properties of the graph [6]. Finding a good representation of patterns may improve the performance of clustering and spectral clustering can give well-performed patterns. Second smallest eigenvalue correspond to the graph cut and its eigenvector embedding the graph into m-dimensional Euclidean space, as a result, it can cluster data with similar patterns [6][8].

2.7 Explain why and how c influences the quality of the clustering

In Part II we use a Gaussian function to calculate the weight matrix since this function measures the similarity between data. Parameter c controls the rapidity of decay of the weight [6] and measures the similarity of points [9]. This function is heavily influenced by the choice of c. When c is small, weight between $\mathbf{x}_i$ and $\mathbf{x}_j$ decays slowly, as a result, it is quite hard to split data with similar or different patterns. On the other hand, when c becomes large, the weight decays very fast, so data becomes very dispersed and it becomes hard to recognize data with similar patterns [10].

3 Part III

3.1 (a)

Here we present the plots depicting the sample complexity of the four algorithms: perceptron, least squares regression, 1-nearest-neighbours and winnow. The vertical axis represents the numbers of samples m, and the horizontal axis represents the dimension of the samples.

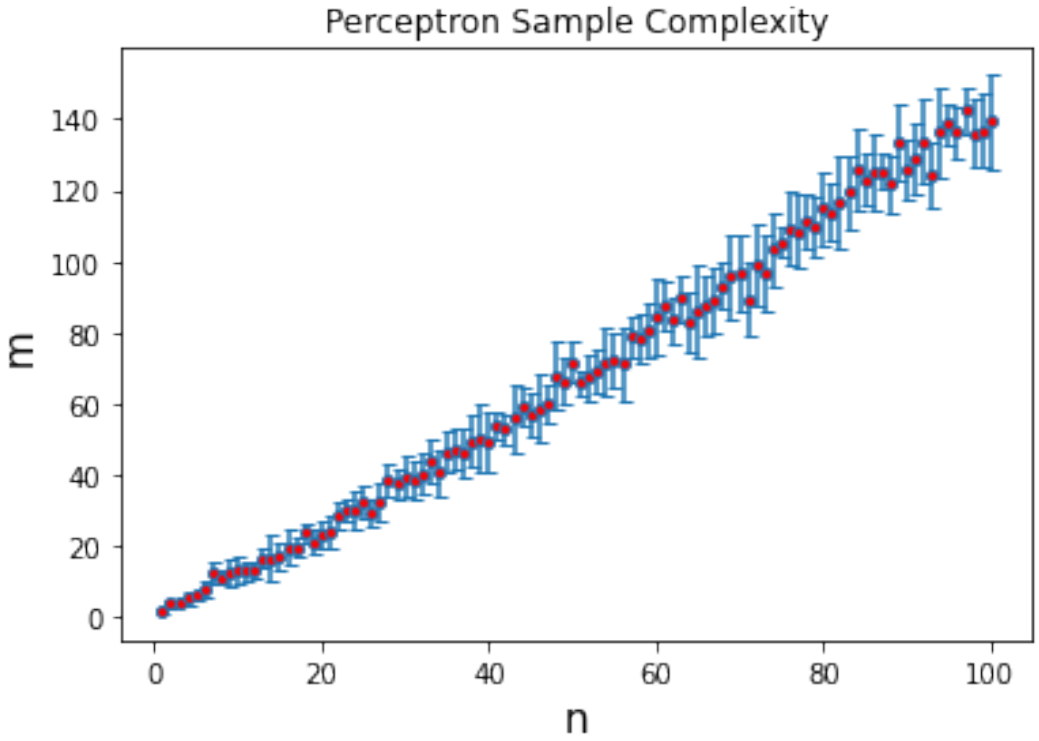

Fig 7 *Sample Complexity for Perceptron*

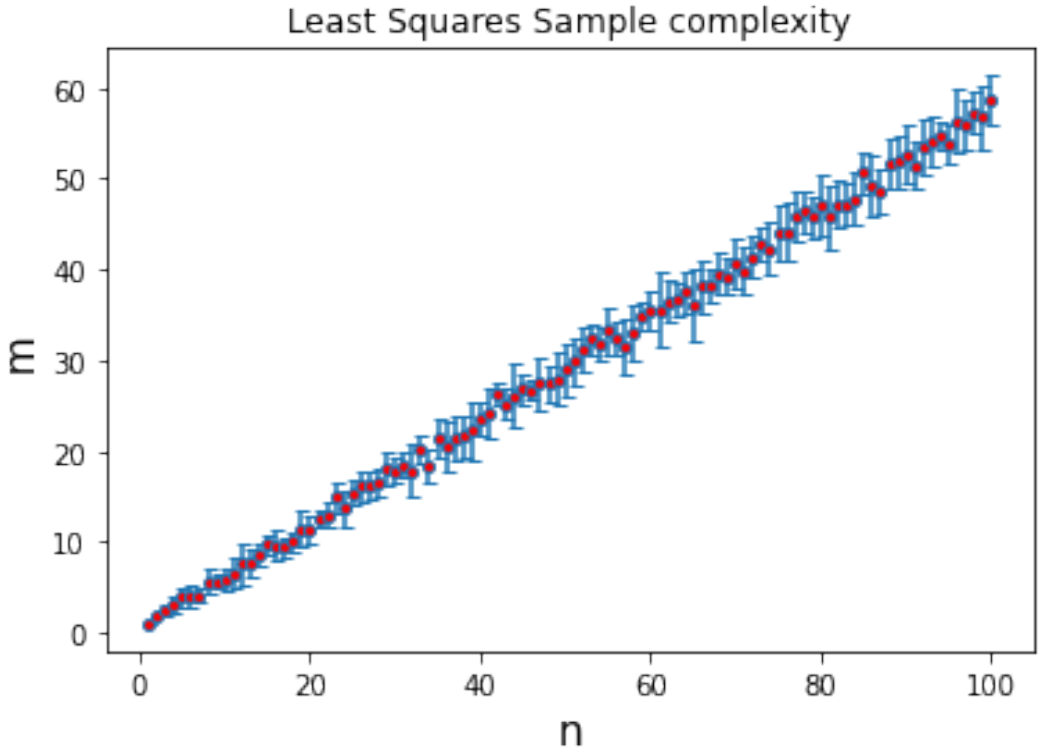

Fig 8 *Sample Complexity for Least Square Regression*

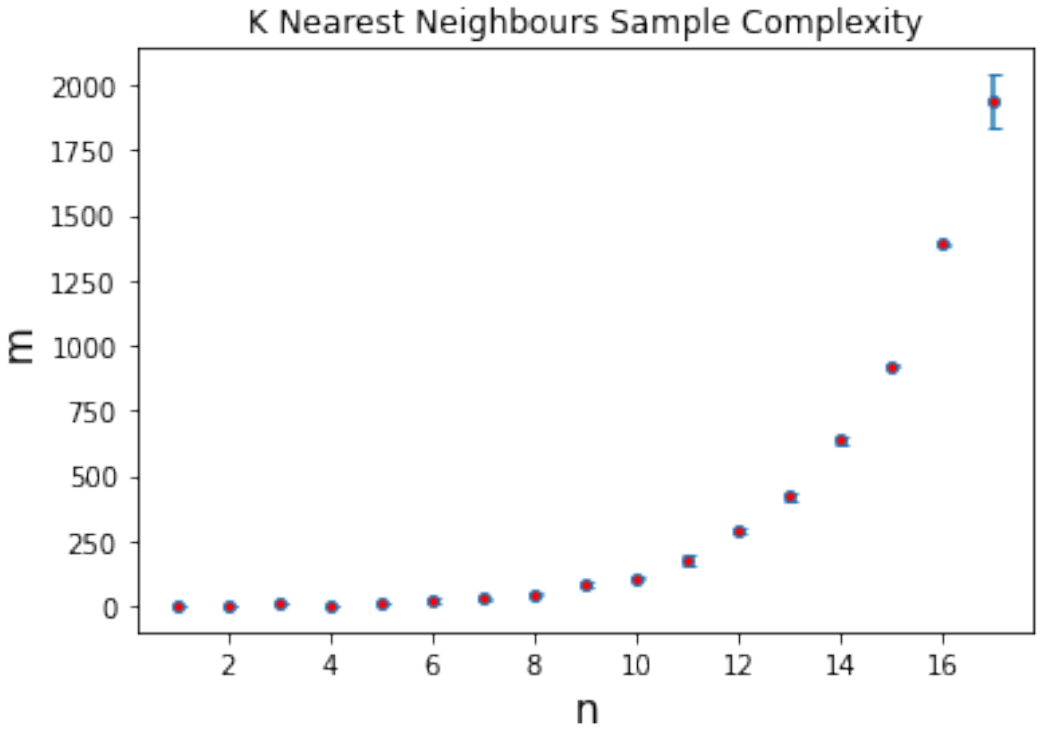

Fig 9 *Sample Complexity for 1-Neareest-Neighbours*

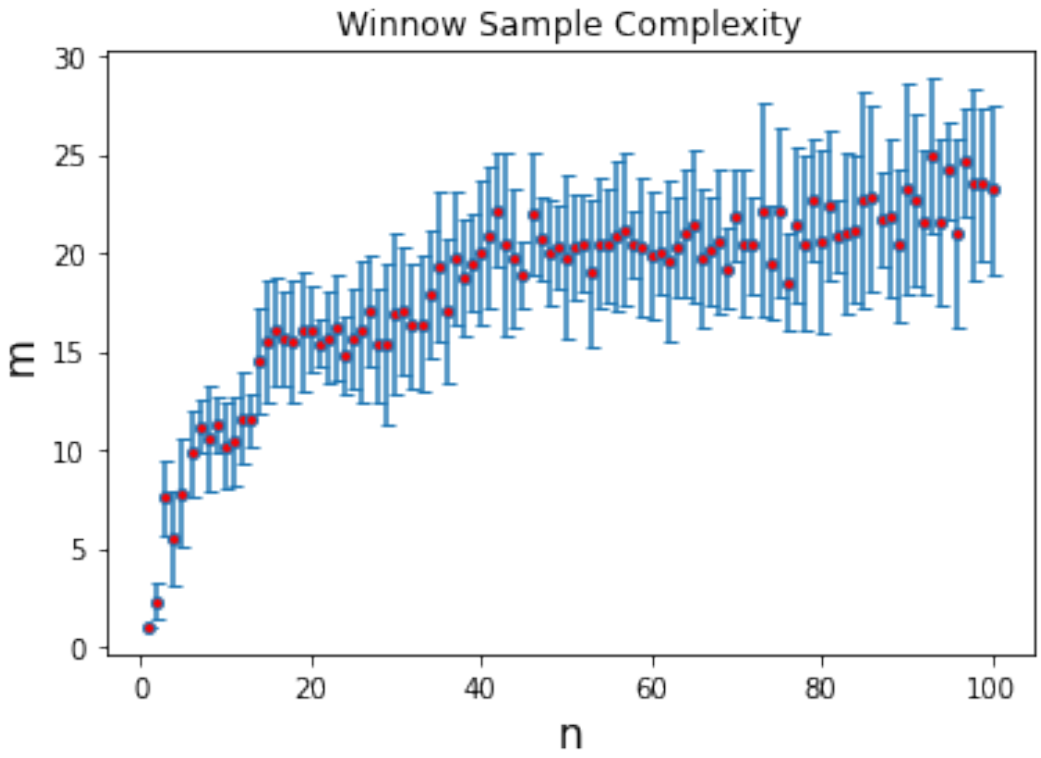

Fig 10 *Sample Complexity for Winnow*

Note that it was not possible to obtain higher dimensions for the k nearest neighbours plot due to the limit of computational resources.

3.2 (b)

3.2.1 Methodology

The method we use is the following (for the analysis of each algorithm):
For each dimension n we train an algorithm with m sample where $m \in \{1, ..., 10000\}$. At each value of m, we test the trained parameters on a dataset consisting of m_{test} $(= 10000)$ points. If the test error is less or equal than 0.1, we record the value of m. We repeat this 20 times and we get 20 m's. We then record the mean value of the 20 m's and this is the sample complexity for a given dimension n. Since we randomly select m_{test} samples from 2^n possible distinct samples, the generalization error is reduced to test error:

$$\frac{1}{m_{test}} \sum_{x \in S_{m_{test}}} I[A_{S_m}(x) \neq x_1]$$

instead of

$$2^{-n} \sum_{x \in \{-1,1\}^n} I[A_{S_m}(x) \neq x_1]$$

For one n this method is $\mathcal{O}(m)$ in m.
A different methodology which can be used is employing a binary search algorithm which for one n is $\mathcal{O}(\log m)$ in m, however due to the first methodology being sufficient we decided not to go with it.

3.2.2 Limitations

A limitation in our method is that we use only 10000 points for our test data, due to the lack of computational resources. 10000 is not even close to the 2^n points we need in order to get the true error. As a result, we gave up accuracy to computational time which is a tradeoff in our experiment. Additionally, due to this limitation, we get a lower value for m and hence our estimates are biased.

3.3 (c)

3.3.1 Estimation

It is clear from the plots above that the perceptron and least squares are increasing linearly, the winnow increasing logarithmically and the 1-nearest neighbour exponentially. To check our hypotheses we fit lines of best fit and get the following:

- Least squares: $\Theta(m(n)) = 0.9418 + 0.6055n \Rightarrow \mathcal{O}(n)$
- Perceptron: $\Theta(m(n)) = -1.2521 + 1.6935n \Rightarrow \mathcal{O}(n)$
- Winnow: $\Theta(m(n)) = 1.2271 + 3.9404 \log n \Rightarrow \mathcal{O}(\log n)$
- 1-Nearest-Neighbours: $\Theta(m(n)) = 0.684 + 0.43 \exp n \Rightarrow \mathcal{O}(\exp n)$

We justify the use of Θ since the estimate of the sample complexity lines are asymptotically bounded below and above by their products with constants as shown in figures 11, 12, 13, 14 below:

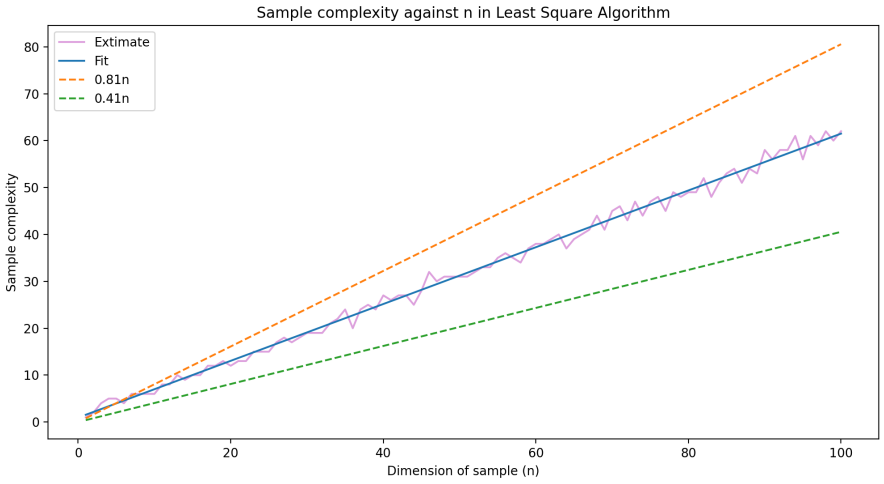

Fig 11 *Least Squares:* $0.9418 + 0.6055n$

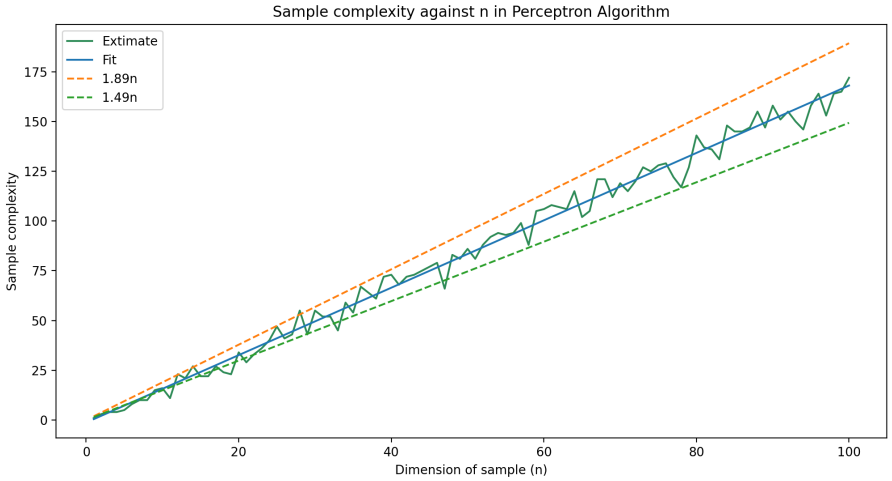

Fig 12 *Perceptron:* $-1.2521 + 1.6935n$

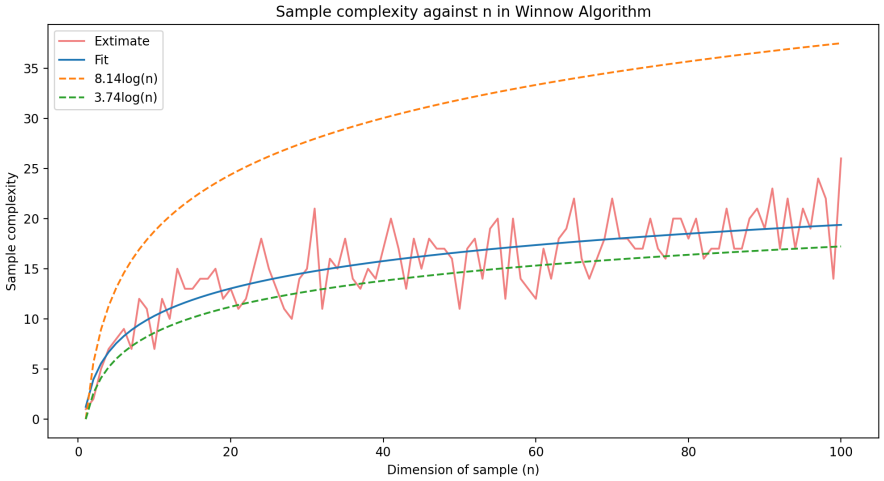

Fig 13 *Winnow:* $1.2271 + 3.9404 \log n$

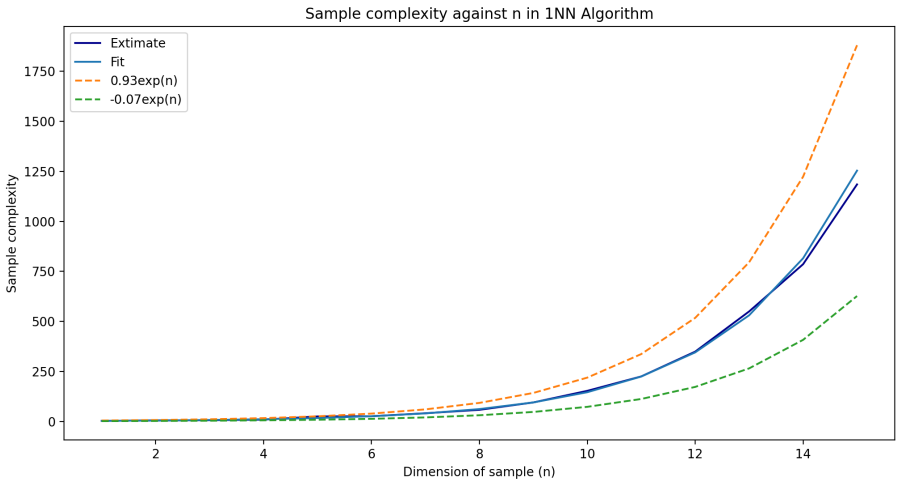

Fig 14 *1-Nearest-Neighbours:* $0.684 + 0.43 \exp n$

3.3.2 Comparison

It is clear that best performing algorithm in terms of sample complexity is winnow, followed by least squares, then perceptron and lastly 1-nearest neighbours. We will now compare them in more detail: We first compare winnow and perceptron as they are similar in nature. It is evident that winnow requires less samples than perceptron to generalize better. To see this, as shown in the lecture notes, the perceptron's and winnow's mistake bounds are $(4k+1)(n+1)$ and $3k(\log n + 1) + 2$ respectively, where k is the number of literals. Using the theorem in the lecture notes (*Online bounds give Batch Bounds*), we know that with a mistake bound B, the error rate for both algorithms is $\frac{B}{m}$
Hence, for the perceptron to have error rate less than 0.1 we require that:

$$\frac{B}{m} \leq 0.1 \iff \frac{(4k+1)(n+1)}{0.1} \leq m$$

where $\frac{(4k+1)(n+1)}{0.1} \in \Omega(n)$
Similary for the winnow algorithm we have that

$$\frac{B}{m} \leq 0.1 \iff \frac{3k(\log n + 1) + 2}{0.1} \leq m$$

where $\frac{3k(\log n+1)+2}{0.1} \in \Omega(\log n)$.
Hence, winnow is clearly better in terms of sample complexity than the perceptron.
Similarly to winnow, least squares requires less samples than the perceptron, although at first glance they seem similar. This is due to the fact that linear regression uses every data point of the training set whilst the perceptron uses 1 point at a time. Hence, for small datasets the perceptron is at a

disadvantage in terms of sample complexity because it requires more samples to converge to a good generalization error.

Lastly, for the 1-nearest-neighbour algorithm due to the curse of dimensionality, the distances between the points, including the test point, increases exponentially. This requires m to grow exponentially to fill in these gaps so that when a test point is compared with the training points, it will have enough confidence that it's the correct class therefore reducing error.

3.4 (d)

Suppose the data is drawn from a distribution P and a mistake bound for algorithm $\mathcal{A}$ for any such set is B. By the theorem in the lecture notes (*Online bounds give Batch Bounds*), the probability of making a mistake on x_s is bounded by $\frac{B}{m}$ i.e.,

$$P(\mathcal{A}(x_s) \neq y_s) \leq \frac{B}{m}$$

. By the perceptron bound theorem (Novikoff), it is known that

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

where $R = \max_t \|x_t\|$ and since $x_t \in \{-1, 1\}^n$, we get that $\sum_i^n x_{t,i}^2 = n \Rightarrow R = \max_t \sqrt{\sum_{i=1}^n x_{t,i}^2} = \sqrt{n}$ where n is the dimension of the samples,. It follows that

$$B \leq \left(\frac{R}{\gamma}\right)^2 \leq \frac{n}{\gamma^2}$$

Therefore

$$\hat{p}_{n,m} = P(\mathcal{A}(x_s) \neq y_s) \leq \frac{n}{m\gamma^2}$$

3.5 (e)

We have that the volume of a Thiessen polygon is $\frac{2^n}{m}$ given the m datapoints $x_i \in \{-1, 1\}^n$ are randomly distributed. Now, the distance of any two points from one another is approximately $\frac{2}{m^{\frac{1}{n}}}$. Let x_{nn} be the nearest neighbour to x. By assumption of the "just a little bit problem", if x_{nn} was to correctly classify x, we require that the first element of x_{nn} is equal to the first element of x. So the probability that x_{nn} correctly classifies x is then $\frac{n-a}{n}$, where a is the number of elements in x_{nn} which are different from the respective elements of x. Hence, $a = \frac{1}{m^{\frac{2}{n}}}$ and $P(\mathcal{A}_{S_m}(x) \neq x_1) = \frac{1}{nm^{\frac{2}{n}}}$. With this, we deduce that for this probability to be smaller than 0.1, the following is required to be true:

$$P(\mathcal{A}_{S_m}(x) \neq x_1) = \frac{1}{nm^{\frac{n}{2}}} \leq 0.1 \Rightarrow m \geq \frac{10 \exp \frac{n}{2}}{n}$$

Now, take general $\epsilon \in (0,1)$ and we get that $m \geq \frac{\exp \frac{n}{2}}{n} \in \Omega\left(\frac{\exp \frac{n}{2}}{n}\right)$

4 References

[1] Pattern Recognition and Machine Learning, Bishop

[2] Vapnik, V. (1998). Statistical Learning Theory.

[3] Shawe-Taylor, J., Cristianini, N. (2004). Kernel methods for pattern analysis. Cambridge university press.

[4] Prechelt, L., Early Stopping - but when?

[5] Li, Q., Song, Y., Zhang, J., Sheng, V. S. (2020). Multiclass imbalanced learning with one-versus-one decomposition and spectral clustering. Expert Systems with Applications, 147, 113152.

[6] Filippone, M., Camastra, F., Masulli, F., Rovetta, S. (2008). A survey of kernel and spectral methods for clustering. Pattern recognition, 41(1), 176-190.

[7] Von Luxburg, U. (2007). A tutorial on spectral clustering. Statistics and computing, 17(4), 395-416.

[8] Belkin, M., Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. Neural computation, 15(6), 1373-1396

[9] Zhang, X., Li, J., Yu, H. (2011). Local density adaptive similarity measurement for spectral clustering. Pattern Recognition Letters, 32(2), 352-358.

[10] Suryanarayana, S. V., Rao, G. V., Swamy, G. V. (2016, August). Role of the scaling factor in spectral clustering algorithm. In 2016 International Conference on Inventive Computation Technologies (ICICT) (Vol. 3, pp. 1-6). IEEE.