COMP0078 Supervised Learning

Coursework 2 Report

Team: Yes

Student ID: 18064052 & 17126493

Department of Computer Science

December 21, 2022

Contents

1 Introduction

This report details and evaluated various Supervised Learning techniques in their ability to model complex data.

2 Part 1: Perceptron MNIST Classification

2.1 Multi-class Kernel Perceptron

2.1.1 Generalisation Algorithm Implementations

This part evaluates and explores performance of various linear classifiers and kernel machines on the MNIST dataset. The MNIST dataset is typically the go-to basic dataset for evaluating image-classification performance of various machine learning models. The dataset comprises of hand-drawn digits ranging from 0 to 9 with a total dataset sample size of 9298 records. Each record represents a flattened 16 x 16 image matrix of greyscaled values between -1 and 1. Our dataset is of the form shown in Equation 1 and examples of the MNIST dataset are shown in Figure 1.

$$\{(\vec{x_1}, y_1), ..., (\vec{x_m}, y_m)\} \in [-1, 1]^n, \{0, 1...9\}^m \tag{1}$$

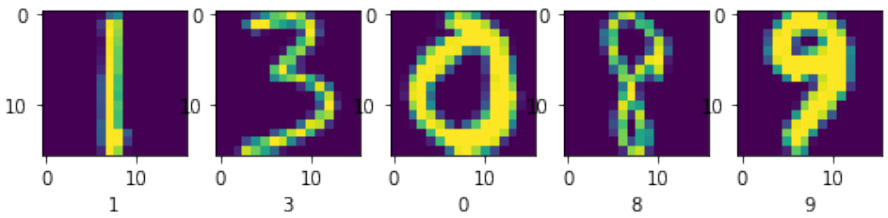

Figure 1: MNIST dataset example images.

We report on 2 variations of the kernelised Perceptron under 2 distinct generalisation settings. The Perceptron is typically a binary classifier but can be generalised to a multi-class setting by reducing the multi-class problem into a set of binary classification problems known as a 'one-versus-rest' approach, where each class is compared to the remaining classes, yielding Perceptron models for each discriminating class [1]. In this paper we compare the standard 'one-versus' approach to an all-rounded 'all-together' approach based on the work of Wetson and Watkins [2] that doesn't consider each class in its own distinct binary case, but defines a single objective function for training every class's Perceptron simultaneously based on a single maximising margin optimisation. Rough sketches of each algorithm's generalisation derived from their respective publications are shown in Figures 2 and 3.

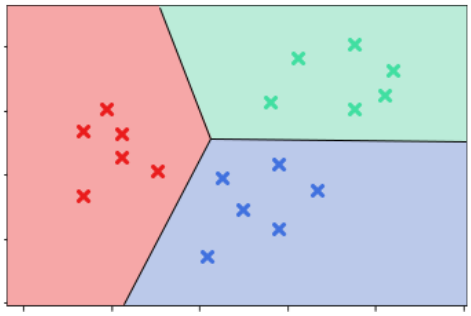

Figure 2: All-together algorithm rough visualisation.

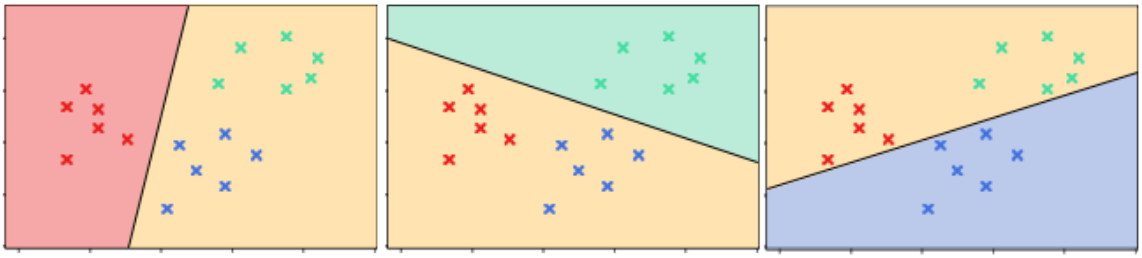

Figure 3: One-vs-rest algorithm rough visualisation.

All-together generalisation approach

Our main generalisation technique on the binary Perceptron is that of [3] via use of the kernel-trick to build an 'all-together' batched version of the online multi-class kernel Perceptron algorithm (MKPA). This method defines kernel-based discriminant functions for each class, $c \in C$, where each discriminant function is defined by all training samples, M, meaning $O(C(M+1))$ parameters to be solved. We then predict class membership by discriminating on a test point with all discriminant functions.

For a class, c, its kernel-based discriminant function is defined as,

$$f_c(\vec{x}) = \sum_{i=1}^{m} \alpha_i^c k(\vec{x}_i, \vec{x}) + \beta_c, c = 0, .., 9 \tag{2}$$

where α_i^c and β_c denote the parameters of class c's discriminant function. The kernel function could be that of any function that satisfies Mercer's condition. For some input vector $\vec{x}$, we say it is classified to the class with the largest discriminant function value, $\hat{y}$, such that:

$$\hat{y} = argmax_{i=0,..,9}(f_i(\vec{x}))$$

For some training sample $\vec{x}_q$ belonging to class c, if there is at least one $c \neq j$ where $f_c(\vec{x}_q) \leq f_j(\vec{x}_q)$, then the vector has been misclassified as its truth discriminant function was not found to be the maximum prediction. We then update our parameters using the following iterative update rule:

$$\alpha_i^c \Leftarrow \alpha_i^c + k(\vec{x}_i, \vec{x}_q), \beta_c \Leftarrow \beta_c + 1 \mid \alpha_i^j \Leftarrow \alpha_i^j - k(\vec{x}_i, \vec{x}_q), \beta_j \Leftarrow \beta_j - 1 \tag{3}$$

The mapped example is added onto the it's own class discriminant function (DC) parameters and subtracted for all classes that predicted with a larger confidence than the true class. In contrast to a 'one-versus' approach, we do not add onto the correct class DC for every larger DC, we only credit it once and simultaneously subtract from all overly-confident misclassfied DCs - a conservative update approach.

One epoch cycles the entire train-set and we therefore completely vectorise this implementation to create a fast batched version, where we can pre-compute a Gram matrix over our entire dataset for dynamic referencing during each stage of the algorithm, taking up $O(M^2)$ space for a decrease in up to $O(CM^2)$ computations in the prediction stage alone. Further optimisations performed looked into decomposing our kernel functions into self inner-products and utilising Numpy's dot-product BLAS optimisations for faster computation of the entire Gram matrix. We therefore have 'fast' versions of each standard kernel function used within this paper. A summary of the 'all-together' algorithm can be found in Table 1.

We expect this approach to perform better than a 'one-versus' approach. 'One-versus' has it's drawbacks in that it treats each class as a binary classification of all samples belonging to itself and all other class samples, meaning large sample imbalances between positive and negative samples for a given class, losing symmetry with the original problem. Using 'all-together', updates occur only at the end of a full training cycle and occur holistically, therefore we expect to encounter less training error instabilities as we are not constantly updating model parameters after each prediction. This does mean, however, this approach will take longer to train due to the single optimisation problem lending an overall cost of $O(C^2N^2)$ [1].

Table 1: All-together kernel Perceptron generalisation algorithm.

	All-together Multi-Class Kernel Perceptron Algorithm
Input	$\{(\vec{x_1}, y_1), ..., (\vec{x_m}, y_m)\} \in \mathbb{R}^n, \{0, 1...9\}^m$
Initialisation	$\vec{\alpha}_{m,c} = 0, \vec{\beta_c} = 0$
Prediction	For a given pattern: $\vec{x}_t \in \mathbb{R}^n$ $\hat{y}_t = max\, f_c(\vec{x}_t) = argmax_c \sum_{i=1}^{m} \alpha_i^c k(\vec{x}_i, \vec{x_t}) + \beta_c, c = 0, 1, ..., C$
Update	$\forall f_c(\vec{x}_t)$ **if** $f_{y_t}(\vec{x}_t) \leq f_c(\vec{x}_t) : \; y_t \neq c$ **then** $\hat{y}_t \neq y_t$: $\alpha_i^{y_t} \Leftarrow \alpha_i^{y_t} + k(\vec{x}_i, \vec{x}_q), \beta_{y_t} \Leftarrow \beta_{y_t} + 1$: $\alpha_i^c \Leftarrow \alpha_i^c - k(\vec{x}_i, \vec{x}_q), \beta_c \Leftarrow \beta_c - 1$

Alternate one-versus-rest generalisation approach

Our alternate algorithm for generalising the kernel Perceptron is a 'one-versus-rest' approach. This approach yields a Perceptron model for each class, where each model for a given class considers its own class training samples as $+1$ labels and all other training samples to be -1 labels. The multi-class problem then decomposes back into a simple binary kernel Perceptron problem. The algorithm for this method is much more straightforward, and so we summarise it entirely in Table 2. This approach solves for $O(CN)$ parameters, 1 less than the 'all-together' approach as we do not keep track of a biasing β_c parameter. The model also trains in an entirely online setting, where model

updates occur on a trial-by-trial basis as each training sample is shown to the model, lending an overall cost of $O(CN^2)$.

Table 2: One-versus-rest kernel Perceptron generalisation algorithm.

	One-vs-rest Multi-Class Kernel Perceptron Algorithm
Input	$\{(\vec{x_1}, y_1), ..., (\vec{x_m}, y_m)\} \in \mathbb{R}^n, \{0, 1...9\}^m$
Initialisation	$\vec{\alpha}_{m,c} = 0$
Prediction	For a given pattern: $\vec{x}_t \in \mathbb{R}^n$ $\hat{y}_t = argmax_c \sum_{i=1}^{m} \alpha_i^c \, y_i \, k(\vec{x}_i, \vec{x}_t), c = 0, 1, ..., C$
Update	**if** $\hat{y}_t \neq y_t$: $\alpha_t^{y_t} \Leftarrow \alpha_t^{y_t} + 1$ $\alpha_t^{\hat{y}_t} \Leftarrow \alpha_t^{\hat{y}_t} - 1$

Preliminary Experimentation and Non-Cross-Validated Parameters
The number of total iterations for the, epochs, to train our model was experimentally set to be around 20 for the 'all-together' model. This is a parameter we did not cross-validate over as the MKPA will always converge given a separable dataset mapping. Figure 4 shows us that 20 is a good enough stopping criterion as we trade-off computation time for total convergence. The 'one-vs-rest' model converges quickly at only around 10 epochs which will act as the stopping criterion. An automatic stopping criterion based on convergence of a moving average on model accuracy/parameters could have been used but numerical instabilities may lead to different epoch counts for each experiment and so we feel there is more of a detrimental impact to experimental results than gain and decided it is best to be consistent across all experiments. An issue could be that the second model has more of a chance to converge which could lead to lower error rates and worse comparisons, however this is a trade-off for faster computation.

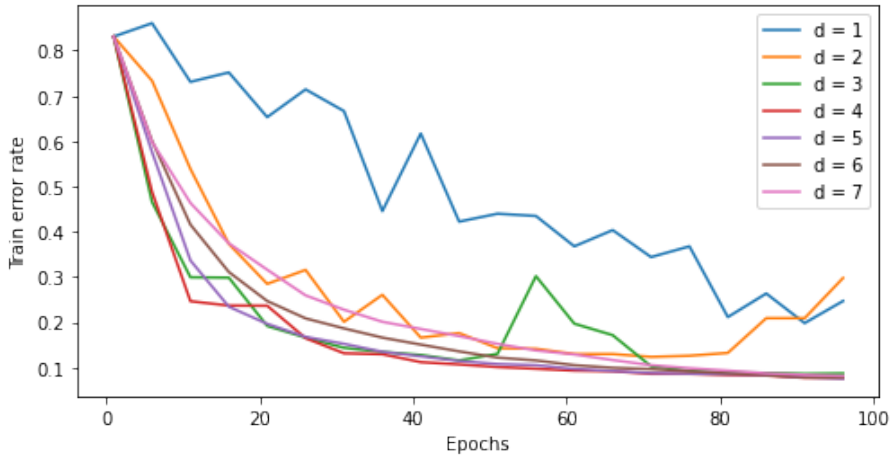

Figure 4: Model training error vs. epoch (Polynomial Kernel All-Together Model).

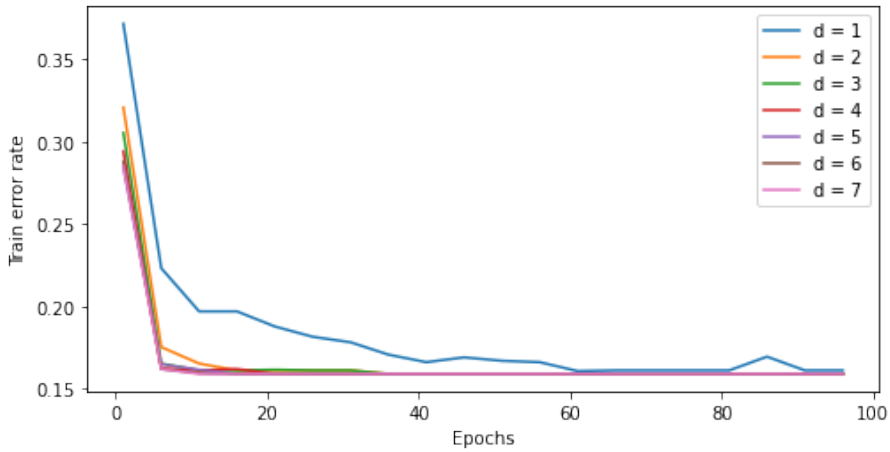

Figure 5: Model training error vs. epoch (Polynomial Kernel One-Vs-Rest Model).

2.1.2 Algorithm Comparison with a Polynomial Kernel

Basic Experimentation to Identify Optimal Polynomial Dimension

The MKPA's are first tested with a Polynomial kernel given in Equation 4. We wish to identify the optimal hyperparameter d*, the dimension of the polynomial kernel, that best fits our dataset, where $d \in 1, 2, ..., 7$, and yields the lowest generalisation test-set error.

$$K_d(\vec{p}, \vec{q}) = (\vec{p} \cdot \vec{q})^d \tag{4}$$

We first conduct a basic approach to identify d* by performing 20 runs for each dimension $1, 2, ..., 7$ where we split our dataset into a random 80% train-set and a 20% test-set for each cycle and dimension. The test and train set errors were measured and the averages given in Table 3.

Table 3: Average train and test error for each polynomial dimension for each algorithm.

	All-Together Algorithm		One-Vs-Rest Algorithm	
d	Mean Train Error	Mean Test Error	Mean Train Error	Mean Test Error
1	0.67±0.048	0.62±0.078	0.22±0.003	0.24±0.011
2	0.47±0.090	0.46±0.090	0.17±0.003	0.20±0.008
3	0.40±0.057	0.35±0.038	0.17±0.002	0.20±0.006
4	0.32±0.011	0.29±0.012	0.17±0.002	0.20±0.007
5	0.17±0.010	0.17±0.010	0.17±0.002	0.20±0.011
6	0.13±0.003	0.14±0.006	0.17±0.002	0.20±0.009
7	0.20±0.016	0.21±0.022	0.17±0.002	0.19±0.010

Table 3 shows that for the all-together algorithm, generally as the dimensions increase to 6, the mean train and test set errors decrease and predictions become more consistent as indicated by the decreasing standard deviation, but beyond that the errors begin to increase. We expect this result as each increasing dimension contains all function class spaces of its predecessors, which should mean dimension 7 should be the best in theory, yet 6 is indicated as the best. This could be due to the fixed size of parametric modelling via

the polynomial kernel, meaning past a certain dimension, our data cannot be any better modelled even if we increase our training points. In contrast, the one-vs-rest algorithm registers consistent train and test set errors for dimensions 2 and above. This looks to be due to total convergence and completed hyperplane separability of our data as expected from preliminary trials (see Figure 5), however, it could be due to the simpler nature of our all-vs-rest Perceptron where decomposing the multi-class problem into simpler binary classification problems limits the benefits of higher dimensional polynomials.

5-Fold Cross-Validation to Identify Optimal Polynomial Dimension

We now move onto a more robust experiment to identify d* and perform cross-validation (CV). CV is more robust as we train and test on multiple subsets, folds, of our training set, therefore decreasing potential bias that could arise from highly coupling our model training and evaluation with one train-test split. We perform 20 runs with an 80%/20% randomly sampled train-test dataset split. The 80% train split is further split into 5 'folds' and each iteration of CV trains a Perceptron model on 4 folds, whilst holding-out the last fold to use as a validation set to measure the validation accuracy. This is repeated with each fold being used as a validation set, and the validation errors are averaged and returned. This is then repeated for each dimension and the lowest erroneous dimension in each cycle is used to train a Perceptron model on the the full 80% train split and tested on the remaining 20% test split. With 5-fold CV, over 20 runs, we will obtain 20 best found polynomial dimensions, d*, for each cycle and their respective test-set errors. Our results are shown in Table 4.

Once again, we get varying results across our two MKPA algorithms. The all-together method identifies the best polynomial dimension to be $d = 6 \pm 0.3$ with a low generalisation error of only 14% and this was found to be very consistent across all cycles. The one-vs-rest method produced a range of best dimensions ranging from 2 to 7, almost covering the entire set of polynomial dimensions, and an average best dimension to be $d = 4 \pm 1.36$ with high uncertainty! This is expected as the results of our previous experiment yielded the same test-set errors across all dimensions. Allowing the Perceptron model to train to convergence clearly diminishes the effect of higher-order polynomial kernels. The all-together algorithm stops before it totally converges and so better dimension polynomials would have converged faster than others, therefore allowing us to evaluate each polynomial dimension better.

We conclude that amongst both approaches, the all-together approach seems to be less susceptible to numerical instabilities, and is overall more robust in that it saturates less at higher order polynomials and reached generalisation errors of 6% lower on average over the all-vs-rest approach. Due to higher reliability of the all-together MPKA results, we also conclude the optimal dimension to be $d = 6$.

Table 4: 5-fold cross-validation: best d* for each cycle and measured test-set error.

	All-Together Algorithm		One-Vs-Rest Algorithm	
Cycle	Best d*	Test Error	Best d*	Test Error
1	7	0.20	6	0.19
2	6	0.13	4	0.19
3	6	0.13	3	0.19
4	6	0.15	5	0.19
5	6	0.13	3	0.21
6	6	0.14	3	0.20
7	6	0.14	7	0.21
8	6	0.12	4	0.20
9	6	0.13	6	0.20
10	6	0.14	4	0.20
11	6	0.14	4	0.19
12	6	0.13	5	0.18
13	6	0.13	5	0.19
14	6	0.13	7	0.19
15	6	0.14	5	0.20
16	6	0.15	3	0.19
17	6	0.14	3	0.21
18	6	0.14	4	0.20
19	6	0.15	2	0.21
20	7	0.13	5	0.20
Mean Over Cycles	6.1±0.3	0.14±0.02	4.4±1.36	0.20±0.01

Confusion Matrix for Cross-Validation Experiment (All-Together MKPA)
Among each of the 20 runs we can visualise which examples were continuously misclassified through a confusion matrix which is shown in Figure 6. The confusion matrix plots the error rate for each class, normalised by the total mistakes over the entire cycle, and this was averaged over all 20 cycles. The highest error class is class 9 in which the model predicts a 4 and this was found to be very high at 9.3% error rate! The digit 9 was also highly mistaken for the digit 7. On a whole, the digit 2 had the highest total error rate across all possible predicted labels.

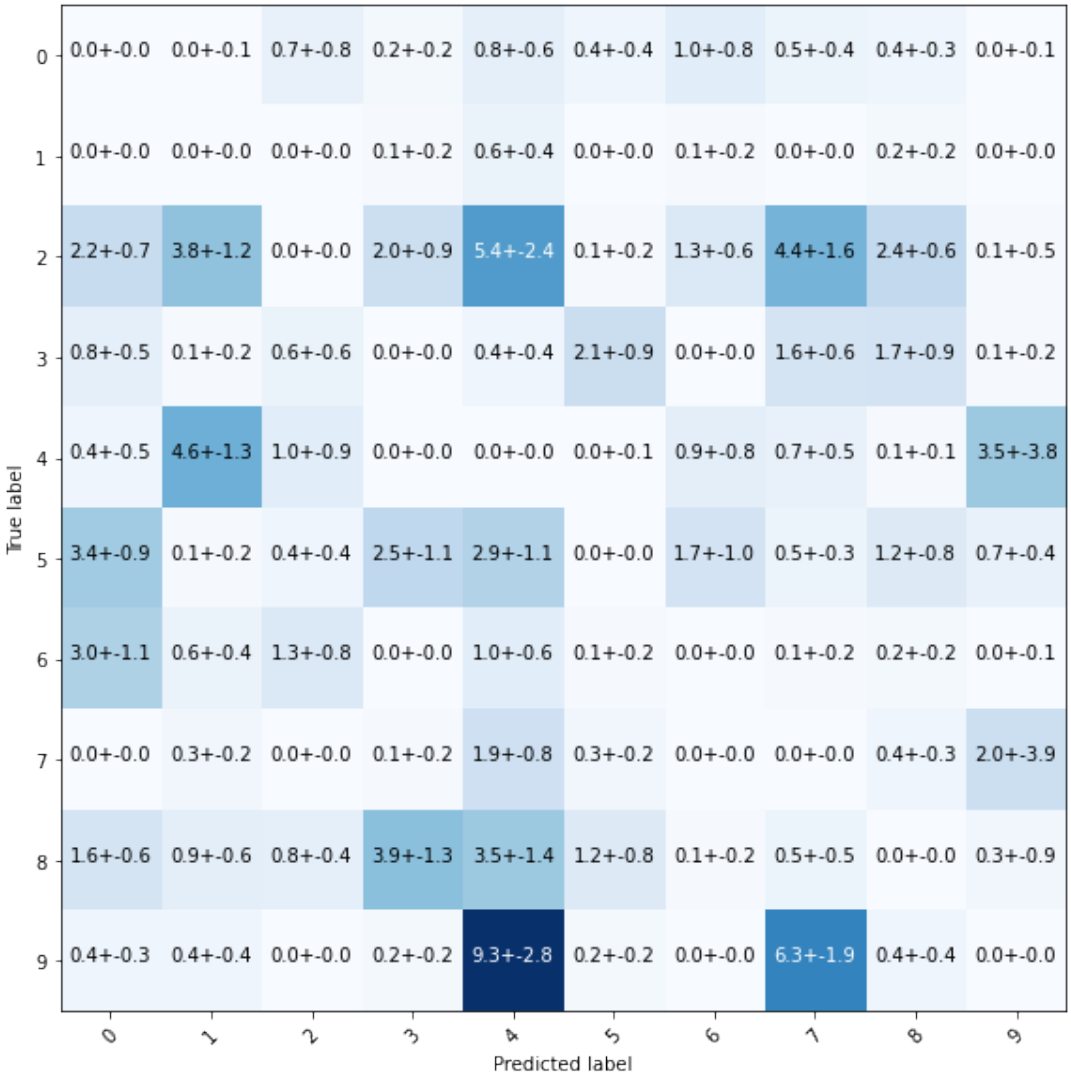

Figure 6: Confusion matrix average error rate over 20 cycles for 'All-Together' algorithm (%).

2.1.3 Top 5 Hardest To Predict Samples

We have a general notion from our cross-validation confusion matrix as to which class of samples should be the hardest to classify. To identify the top 5 hardest, we train 50 Perceptron models with random 80/20 data set splits using the Polynomial kernel with dimension 6, as was identified to be the best dimension though earlier experiments. We then use each model to predict over the entire dataset and use a hashmap to keep track of the total number of mistakes made for each data point. The 5 hardest labels are shown in descending order of difficulty to predict in Figure 7.

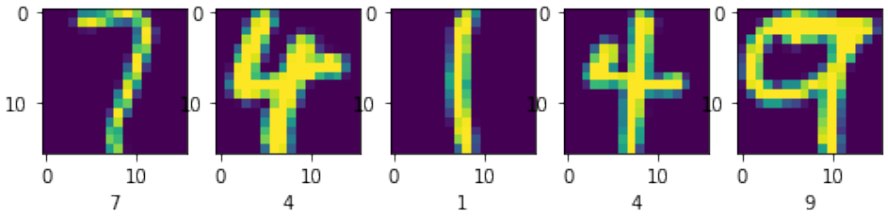

Figure 7: Top 5 hardest samples to classify.

We see 3 of our hardest predictions consist of two 4's and a 9 which is expected as these values have significantly higher confusion error rates in Figure 6. What is surprising is that the hardest to predict image belonged to class 7 and the third hardest belongs to class 1, which both had some of the lowest confusion rates. This could be because the 7 is very pixelated with little maximums (+1's) within the main body of the digit and due to its shape being similar to the number 9.

2.1.4 All-Together MKPA with a Gaussian Kernel

We now wish to compare the use of MKPA with a radial-basis function kernel. The same basic results and cross-validation experiment as detailed in Section 2.1.2 was performed. We use the Gaussian kernel instead (Equation 5 and wish to identify the best width parameter, $c = \frac{1}{2\sigma^2}$, that maximises model accuracy. A k-order Polynomial kernel is limited in that it only represents all analytic functions with a $(k+1)$th order constant derivative and by extension all functions of $(k+2)$th order and above will be zero. The Gaussian kernel gives access to all analytic functions as it is infinitely differentiable and so it should be able to model very complex data better than the Polynomial kernel.

$$K(\vec{p}, \vec{q}) = e^{-c\|\vec{p}-\vec{q}\|} \tag{5}$$

We run initial experimentation to determine the set $S : c \in S$ to experiment over. To do this we scaled c by a factor 10 from 10^{-5} to 10^{5} and trained models on each width to evaluate which range of values performed the best. The optimal identified range was between $\sigma = 4 \rightarrow 7$ and subsequently set our parameter set $S = \{0.01, 0.012, ..., 0.022\}$ where $c \in S$. Our experimentation results of the all-together MKPA over this parameter set are given in Table 5 and 6.

Table 5: Average train and test error for each Gaussian width, c.

	All-Together Algorithm	
c	Mean Train Error	Mean Test Error
0.010	0.34±0.031	0.33±0.031
0.012	0.18±0.033	0.16±0.028
0.014	0.11±0.002	0.11±0.007
0.016	0.09±0.002	0.10±0.006
0.018	0.08±0.003	0.09±0.007
0.020	0.07±0.001	0.08±0.005
0.022	0.07±0.001	0.09±0.006

Table 6: 5-fold cross-validation: optimal kernel width, c*, for each cycle and measured test-set error (Gaussian Kernel using All-Together MKPA).

	All-Together Algorithm	
Cycle	Best c*	Test Error
1	0.020	0.12
2	0.022	0.12
3	0.020	0.09
4	0.20	0.10
5	0.018	0.10
6	0.022	0.12
7	0.022	0.13
8	0.020	0.10
9	0.018	0.10
10	0.018	0.10
11	0.018	0.11
12	0.018	0.12
13	0.022	0.10
14	0.020	0.09
15	0.018	0.09
16	0.018	0.10
17	0.020	0.12
18	0.018	0.11
19	0.020	0.10
20	0.020	0.11
Mean Over Cycles	0.020±0.007	0.11±0.01

Table 5 shows a decreasing trend in train-test errors with the optimal Gaussian kernel width identified at 0.020, which registered a much lower test-set error of 8%, 6% lower than that of the best Polynomial kernel setting. Cross-validation was used once again to corroborate the results of the initial experiment, and the best average optimal Gaussian width was found to be 0.020 ($\sigma = 5$) again. The train and test errors presented a small standard deviation meaning they were very consistent across all cycles. The average test error over CV for the Gaussian kernel was 11%, whereas the Polynomial kernel has a CV test error of 14%.

Both experiments conclude that the Gaussian kernel performs better than the Polynomial kernel on our MNIST dataset which was expected. As we stated previously the Gaussian can model an infinite amount of analytic function spaces and so can model complex data more effectively. The trade-off is that the Gaussian kernel takes more computation to perform, and so for very large datasets it could potentially mean longer training times in which the Polynomial kernel set to a high degree would be a better choice.

3 Part 2: Spectral Clustering

3.1 Implementation of Spectral Clustering

Three different datasets were used in this experiment. Two datasets were predefined and named as dtrain123.dat and twomoons.dat. We built a data generator that creates gaussian distributed 2d data points based on the means and standard deviations. The data generator also checks the covariance matrix of the data points to minimize outliers.

$$W = \left(e^{-c\|x_i - x_j\|}\right)_{i,j=1}^{l} \tag{6}$$

To perform spectral clustering, first we create a weight matrix, W, based on our data matrix using Equation 6. Vectorize the function to achieve maximum running speed. Next, we need to get the graph Laplacian, L. Create a diagonal matrix D, where $D_{i,i}$ is the sum of the ith row of W. Subtract D with W to get L. Finally, calculate the eigenvalues and the eigenvectors. Find the second smallest eigenvalue, and the corresponding eigenvectors are the predictions that we are looking for. Since the label of our data points are {1, -1}, change all positive eigenvectors to 1 and all negative eigenvectors to -1 and that will be the final prediction of spectral clustering.

3.2 Results of the Experiment

All the accuracy of the prediction is calculated using the equation given in the coursework file, which is shown in Equation 7. l_+ is the sum of correctly predicted samples and l_- is the sum of wrongly predicted samples. Basically, we take the maximum of $\{l_-, l_+\}$ and divided by the total samples to get the final prediction accuracy. The reason why this method works is explained later.

$$\mathrm{CP}(c) := \frac{\max\{l_-, l_+\}}{l} \tag{7}$$

3.2.1 twomoon Dataset

The c value range given in the coursework is not large enough to reach the optimal performance (more details in the questions section). We expanded the range to $\{-10.0, -9.9, -9.8, ..., 39.8, 39.9, 40.0\}$ and the optimal c values roughly happen in range 20 - 25. The clustering result and the accuracy across c values are plotted below.

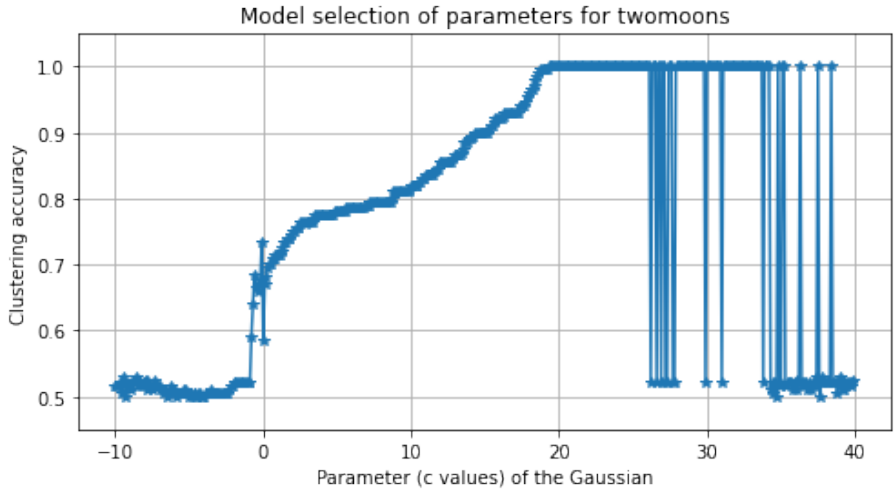

Figure 8: The accuracy across c values for twomoon dataset

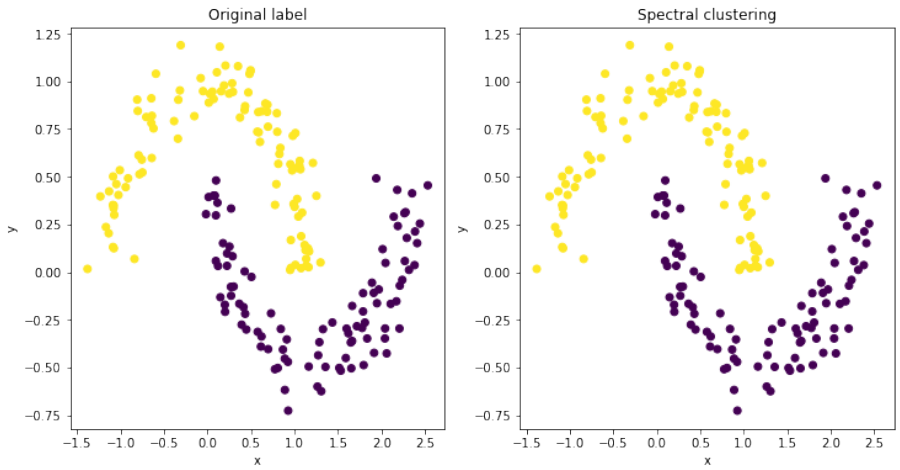

Figure 9: The best clustering result of twomoon dataset

3.2.2 Self Generated Dataset

The covariance check in the data generate section minimized the possibility of outliers. The quality of the clustering highly depends on the separation of the generated dataset, so it is hard to say a fixed c value will produce the best prediction (usually at c = 5 we get accuracy $> 90\%$). It is worth mentioning that the data points very close to the boundary is more likely to be predicted as the cluster centred (0.15, 0.15), because the standard deviation of this cluster is smaller. Smaller standard deviation means the data distribution is more focused, so from the view of datapoints near the boundary, most of the "nearby" datapoints come from the (0.15, 0.15) centred distribution. In figure 10 we see that one datapoint generated from the centre (-0.3, -0.3) is very close to the other clustering group, so inevitably that datapoint will be misclassified (the best result on this example is 97.5% accuracy at c = 16).

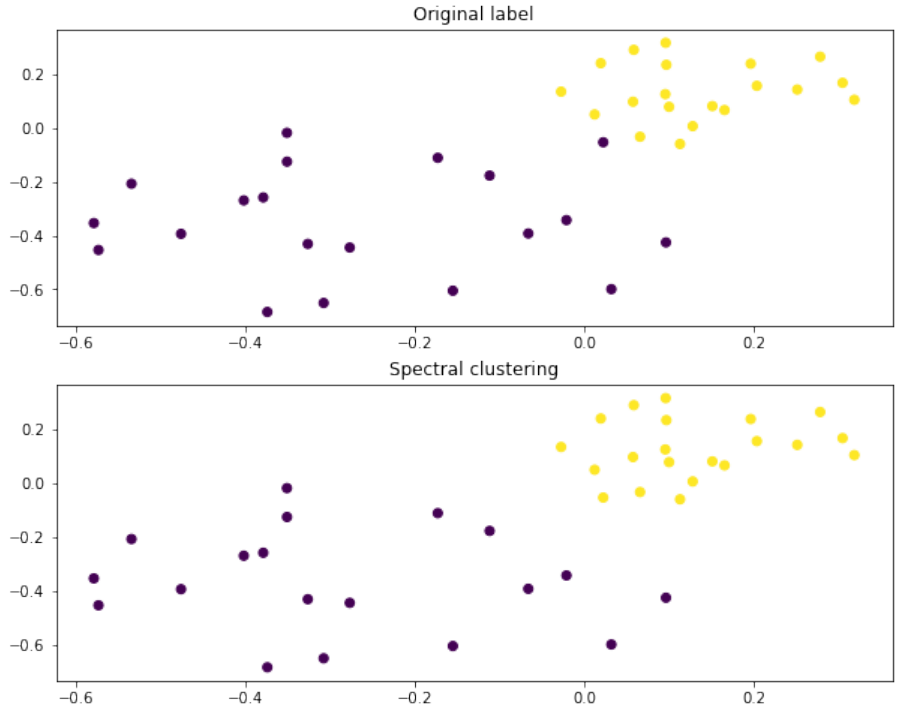

Figure 10: One example of the best clustering result of self-generated dataset

3.2.3 dtrain123 Dataset

This time the best prediction accuracy is 93.2% which happens at c = {0.04, 0.041, 0.052}. The trend in figure 11 is similar to the figure given in the coursework file.

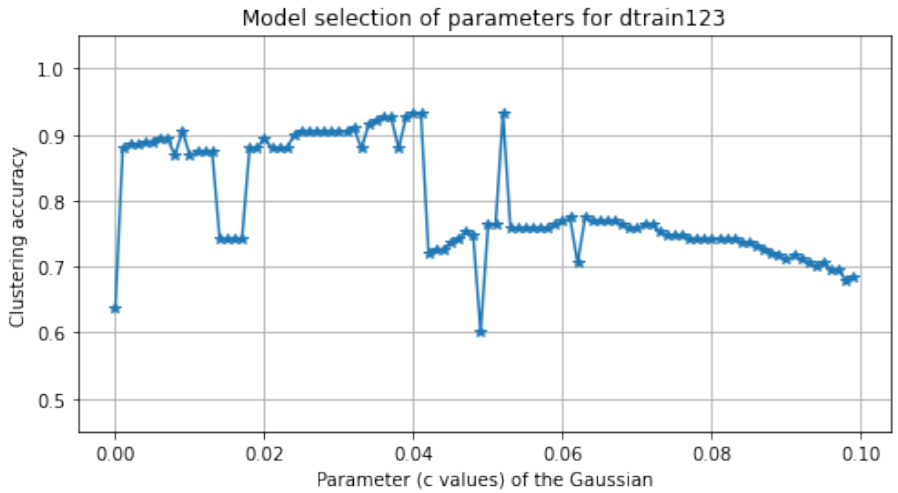

Figure 11: The accuracy across c values for dtrain123 dataset

3.3 Answers to Questions

3.3.1 Explanation of Why CP(c) is a Reasonable Measure

CP(c) is a reasonable method for binary clustering cases. Since there are only two classes to cluster, the lowest accuracy is 50%. If the correct prediction number is lower than

the wrong prediction number, we can simply reverse the two labels and the accuracy will be over 50%. In other words, if a model has an initial correct rate of 0%, it is actually performing excellent clustering but with the opposite labels.

3.3.2 Explanation of Eigenvalue and Eigenvector

The Laplacian is positive semi-definite as given in the lecture notes. Also, we know that the eigenvalues, λ, have the property: $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_l$ Now we consider the relation of the rows of the Laplacian. Notice that the Laplacian is created by $D - W$ and D is created by summing the rows of W. Each row of matrix W represents the relation of one point compared to all other points. If all the points are fully connected, the sum of all the rows will be zero as all the corresponding weights cancel out (summing all rows is similar to Gaussian elimination). This implies that the sum of all rows of the Laplacian will also be zero, meaning that it is not a full rank matrix, so there should be at least one eigenvalue equals to zero. Since the first eigenvalue is the smallest, it must be zero.

Furthermore, assume that v_1 is the eigenvectors of eigenvalue λ_1, we now know that $\lambda_1 = \mathrm{v}_1^{\mathrm{T}} L \mathrm{v}_1 = 0$. As mentioned before the sum of all rows of Laplacian is zero, it is obvious that v_1 is just a constant as zero times any constant is still zero.

3.3.3 Explanation of Why Spectral Clustering Works

Spectral clustering measures the "similarity" between all points. Therefore, each datapoint will have weights to all other datapoints. Higher weights represent stronger relations, and all we need to do is set a threshold that tells whether two datapoints should be in the same clustering or not [4]. In other words, the weights between the edges of different clusters will be relatively lower than the weights within a cluster. This property allows spacial clustering to not assuming any specific shape or distribution, which make this algorithm very adaptable.

3.3.4 Explanation of the Width Parameter c

The parameter c controls the gaussian width, meaning that it decides how far apart two datapoints are still consider as the same cluster. So, for clustering problems that have clear boundaries, larger c gives a more accurate prediction as datapoints from different clustering groups are relatively far away compared to the distance within the clustering group (refer to twomoon example, the ideal c values are mostly in range 20-25). Yet this does not mean the larger the c value, the better the result will be. Because the consequence of very large c value is that the weight matrix will mostly be 0 regardless of the distance between two points. For more complex boundary problems, the tolerance level of large c value are too high, so many irrelevant datapoints will be classified as the same cluster which lowers the performance (refer to dtrain123 example).

4 Part 3: Sparse Learning

4.1 Estimating Sample Complexity and Trade-Off Discussion

The method of estimating sample complexity is shown in the pseudo code below. The parameters that are related to the trade-off are *thresh*, *step*, and *nData*.

Algorithm 1: Estimating sample complexity

```
Data: Algorithm type
Result: Sample complexity
dim ← Maximum dimension;
thresh ← Threshold in percentage;
testing_rounds ← Number of testing rounds;
error_rate ← Results with error > 10% divided by testing_rounds;
step ← The increment value for the dimension range;
nData ← numberofsamples;
for {1, 1 + step, 1 + 2 × step, ..., dim} ;          /* step is a trade-off */
 do
    while error_rate ≤ thresh ;                      /* thresh is a trade-off */
     do
        increment nData ;                  /* increment value is a trade-off */
        for {1, 2, 3, ..., testing_rounds} do
            generate train, test data
            train and predict
            count how many times generalization error is ⩽ 0.1
        end
        calculate error_rate
    end
end
```

Obviously setting *thresh* to 1 will produce the "exact" sample complexity, but will be computational heavy and unnecessary if we are just looking for the trend. We decided our *thresh* values by experimenting, and the bias is that the number of samples will be lower than the true result (trend remains the same).

Also, since we are looking for the trend only, there in no need to examine every single dimension in the range. There is a trade-off between smoother trend line and computational time.

The increment of *nData* is also critical. It is intuitively to set it to 1, but notice that setting it to 2 will decrease the computing time by 50% and the bias is nearly neglectable. However, the method we used is tailored to fit our situation. The increment is small for small *nData* to suite the linear and logarithmic trend, and the increment starts to increase according to the growing *nData* to faster compute the exponential trend.

For a quick estimation, setting *step* to 5 (five times faster) and the increment of *nData* to 2 (two times faster) has already made the computation ten times faster.

4.2 Estimating Sample Complexity for Linear Regression, Perceptron, 1-Nearest-Neighbour and Winnow Classifiers

In this experiment we look to estimate the sample complexity of the Perceptron, Winnow, Least-squares Linear Regression and 1-Nearest-Neighbours algorithms. Each classifier was implemented and its sample complexity estimated using the method outlined in Section 4.1. Figures 12, 13, 14 and 15 plot our estimates of how the number of points, m, required to achieve generalisation error of 10% scales against the dimension of our datums, n. Each plot also features fitted traces for various popular $\Theta()$ trends such as linear, logarithmic and exponential increases in the y value against the x value to aid us in categorising the relationship of how m grows as a function of n as $n \to \infty$. The closest relationship trend of our estimate to the traces is taken to be the trend and these are categorised in Table 7.

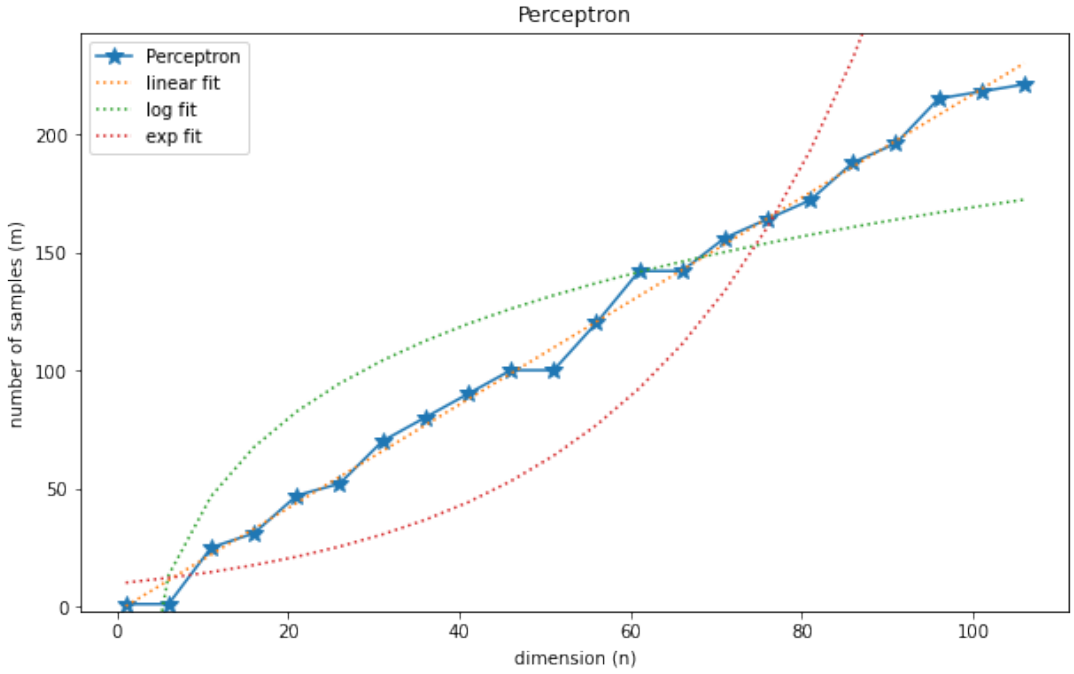

Figure 12: Plot of sample complexity for Perceptron alorithm.

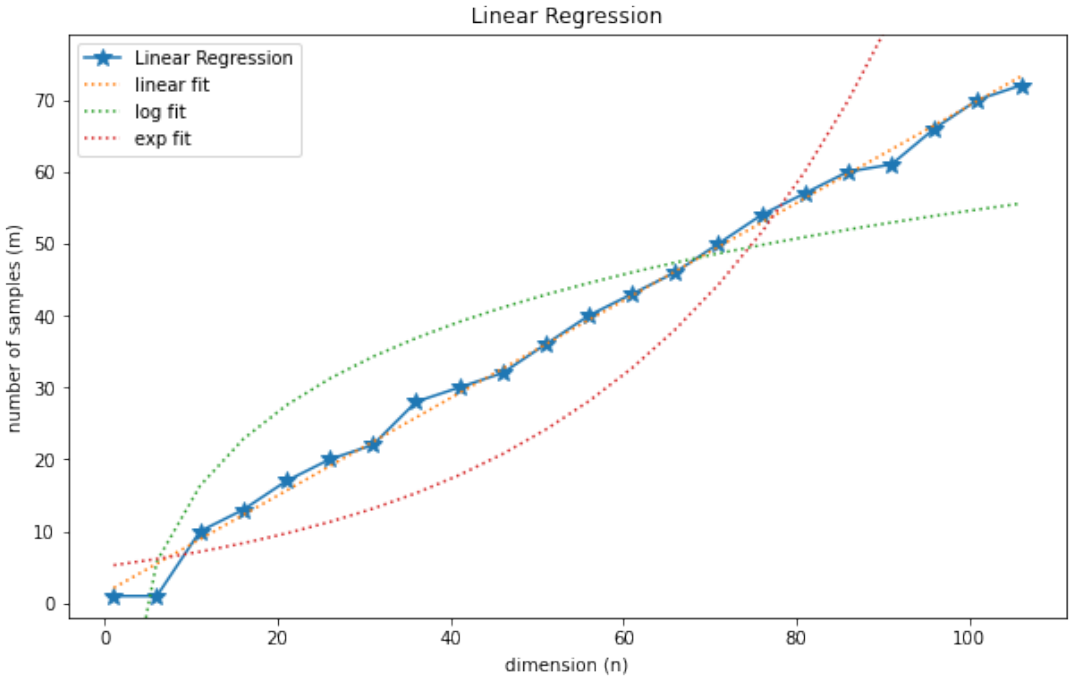

Figure 13: Plot of sample complexity for Linear Regression alorithm.

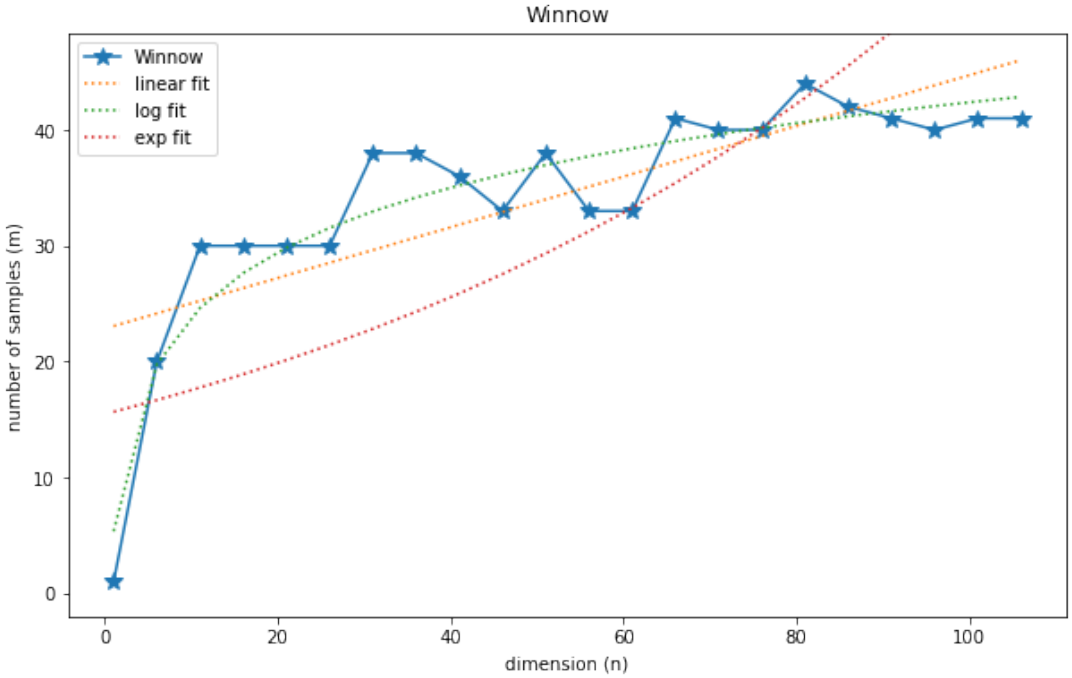

Figure 14: Plot of sample complexity for Winnow alorithm.

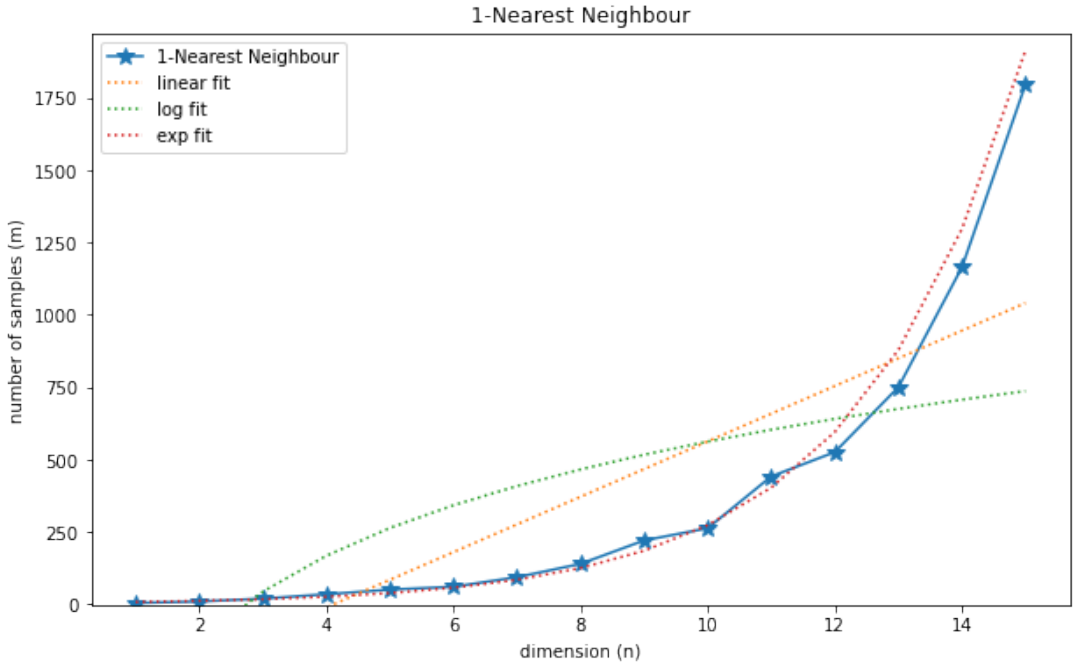

Figure 15: Plot of sample complexity for 1-NN alorithm.

Table 7: Estimated sample complexity for each algorithm.

Algorithm	Relationship of m wrt n
Perceptron	$\Theta(n)$
Linear Regression	$\Theta(n)$
Winnow	$\Theta(log(n))$
1-NN	$\Theta(e^n)$

The bounds from Table 7 indicated that as $n \to \infty$ the algorithm with the best sample complexity looks to be Winnow as it saturates at large orders of n due to it's good DNF mistake bound of $O(kln(n))$ that scales logarithmically with its inputs meaning probabilistically to reach the same error rate the number of required samples scales too. We also note that the Perceptron and linear regression algorithms both have a linear sample complexity but the Perceptron scales faster than linear regression. The Perceptron has a input space of size $R = \sqrt{n}$ and so its mistake bound $\frac{n}{\gamma^2}$ scales linearly with the input dimension where as linear regression scales tightly with its input dimensions as to construct a line (or hyperplane) we need at least as many points as there are dimensions to work with.

1-NN was found to have the worst sample complexity with an exponentially increasing bound. This makes sense as 1-NN is susceptible to the curse of dimensionality, meaning for larger dimensional data, the sample space increases drastically, such that a nearest neighbour in n-D space is much "further" away than it would be at lower dimensions, and so we need to populate the subspace with exponentially more samples for better predictions. Regression and Perceptron give tight bound estimates yet 1-NN bound degenerates beyond

the 12th dimension, and Winnow gives a loose bound which is attributed to our trade-offs for computation time within our estimator function. We still use Θ notation as for each respective algorithm the lower and upper bounds still follow their respective trends.

4.3 Upper-bound Probability of Perceptron Classifier Making a Mistake

Assume that we have examples $(\vec{x}_1, y_1), ..., (\vec{x}_{s-1}, y_{s-1}) \in \{-1, 1\}^n \times \{-1, 1\}$ where s is sampled uniformly at random and $s \in 1, ..., m$. From the lecture slides, the maximum number of mistakes, M, that a perceptron algorithm can make is proven to be,

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

Where $R := max_t \|\vec{x}_t\|$ and γ is a constant. Since our randomly generated data patterns are sampled from $\{-1, 1\}$, we know that,

$$\|\vec{x}_t\| = \sqrt{\vec{x}_{t_1}^2 + ... + \vec{x}_{t_n}^2} = \sqrt{1 + ... + 1} = \sqrt{n}$$

$$\therefore M \leq \frac{n}{\gamma^2}$$

Recall that online sequence of data usually has no distributional assumptions, meaning that $(\vec{x}_s, y_s)$ and $(\vec{x}_{s+1}, y_{s+1})$ are totally independent. A critical outcome is that no matter how many examples the algorithm has trained on, the possibility of encountering an error in the next prediction remains the same. Thus, the probability of producing an error is constant throughout the whole training cycle. The upper-bound on the probability, $\hat{p}_{m,n}$, of the binary Perceptron classifier making a mistake will simply be the average of M, which can be written as Equation 8.

$$\hat{p}_{m,n} = \frac{M}{m} \leq \frac{n}{m\gamma^2} \tag{8}$$

4.4 Challenge: Finding a lower-bound on 1-NN sample complexity

We can find a good lower bound of the sample complexity of an arbritray learner $\chi()$ for the 'just a little bit' problem through computational learning theory. Our problem is formulated over boolean values that are sampled at random from $\{-1, 1\}^n$ with derived labels equal to the first dimension value of the datum. Let us denote a learner, χ_S trained on a set S_m of m patterns drawn uniformly at random and $\chi_S(\vec{x})$ be the prediction of such a learner on pattern $\vec{x}$. We wish to estimate a good bound on the sample complexity such that the generalisation error of our learner for the 'just a little bit' problem is on average 10%:

$$\varepsilon(\chi_S) := 2{-}n \sum_{\vec{x}_{test} \in \{-1,1\}^n} I\left[\chi_S(\vec{x} \neq \vec{x}_1)\right] \tag{9}$$

Given a hypothesis space H with which we can draw learners for the base case, we denote good hypothesis to be those that predict our expected conjunction with at most 10% error where we know our train and test samples are both $\in S$, then

$$H_{bad} = \{h \in H \mid E\left[\varepsilon(\chi_S(\vec{x}))\right] > 0.1\}$$

$$H_{good} = \{h \in H \mid E\left[\varepsilon(\chi_S(\vec{x}))\right] \leq 0.1\}$$

and for our boolean input space of dimension n we can derive that $\|H\| = 2^{2^n}$ as for each dimension, n, we introduce 2^n hypothesis that a learner could use to predict on a given input. An example is given below for $n = 1$ we have 4 hypothesis that predict differently with different generalisation errors generated from Equation 9.

Table 8: Example of possible hypothesis for 1-D input.

Input (d=1)	h1	h2	h3	h4
-1	-1	-1	1	1
1	1	-1	1	-1
Error	0	0.5	0.5	1

However, in the 'just a little bit' case, not all hypothesis are valid. For example, h4 is impossible to reach as it would require training samples of opposite signs co-existing. In fact, we can approximate the number of realisable hypothesis, C, to be $H_C \approx 2^{2^n-1} + 1$. Considering the VC dimension of our problem, for the given example it is possible to see the VC dim is $VC_d = log_2\left(2^{2^n-1} + 1\right) \leq 2^n$, in which we relax the equality and bound to obtain a valid integer dimension. Using Equation 10. C1 is an arbitrary constant that can be set, however the bound still holds for a 'good' C1.

$$m \geq C1\frac{VC_d - 1}{\varepsilon} = C1\frac{log_2\left(2^{2^n-1} + 1\right) - 1}{\varepsilon} \approx C1\frac{2^n - 1}{0.1} \tag{10}$$

Our goal is to find a good lower bound. In other words, what is the minimum samples needed to correctly predict any given point from the distribution. In the 'just a little bit' problem, the label of every data x_i is simply $x_{i,1}$. This means that in any dimension, the two different labels $\{1, -1\}$ will cluster on two symmetric hyperplanes. Therefore, we can find two critical points x_{p1} and x_{p2} that are the closest points with different labels, ie. $x_{p1,1} = -x_{p2,1}$ and $x_{p1,(2,3,\ldots,n)} = x_{p2,(2,3,\ldots,n)}$. Assume all points x_{i_p1} are in the same hyperplane of x_{p1}, then the distance between x_{i_p1} to x_{p1} will be smaller than the distance between x_{i_p1} to x_{p2}, hence will be correctly predicted by 1-NN. Since this apply to all hyperplane in any dimension, we conclude that,

$$m = \Omega(f(n))$$

Where

$$f(n) = 2$$

We also include the code that verifies this result (challenge_e.ipynb).

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[2] Jason Weston and Christopher Watkins. In *Support Vector Machines for Multi-Class Pattern Recognition*, pages 219–224, 01 1999.

[3] Jianhua Xu and Xuegong Zhang. A multiclass kernel perceptron algorithm. In *2005 International Conference on Neural Networks and Brain*, volume 2, pages 717–721, 2005. doi: 10.1109/ICNNB.2005.1614728.

[4] U von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17: 395–416, 2007. doi: https://doi.org/10.1007/s11222-007-9033-z.