Supervised Learning CW2

Marco Carobene, Agnieszka Dobrowolska

January 10, 2020

1 PART I

Handwritten Digits Dataset
The dataset considered in this project consists of 9298 greyscale images of handwritten digits, $\{0, 1, ..., 9\}$, each measuring 16×16 pixels and stored as a vector of 1×256. In this project various classification algorithms will be trained to assign a given image to one of the 10 classes, and their performance will be assessed based on their ability to generalise as well as their time complexity.

1.1 Multi-class Kernel Perceptron for Handwritten Digit Classification

Mathematical Formulation of the Perceptron
The perceptron, first formulated by Rosenblatt in 1958, is a supervised learning model which can be used to perform classification tasks. In it's most primitive form, the perceptron constructs linear hyperplanes in $\mathbb{R}^n$ to divide observations $(\mathbf{x}_1, y_1), ...(\mathbf{x}_m, y_m),\ x_i \in \mathbb{R}^n,\ y_i \in \{-1, 1\}$ into binary classes. The hyperplane can be defined as

$$\mathbf{f}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = 0 \tag{1}$$

where $\mathbf{w} \in \mathbb{R}^n,\ \mathbf{x} \in \mathbb{R}^{n \times m}$ and the bias term has been omitted.

The prediction is made such that

$$\hat{y}_i = \text{sign}(\mathbf{w}^T \mathbf{x_i}) \tag{2}$$

$$\hat{y}_i = \begin{cases} -1; & \mathbf{w}^T \mathbf{x_i} < 0 \\ +1; & \mathbf{w}^T \mathbf{x_i} > 0 \end{cases}$$

The goal is to find a set of weights which minimises the number of misclassifications, $\mathcal{D}(\mathbf{w})$:

$$\mathcal{D}(\mathbf{w}) = -\sum_{i \in \mathcal{M}} y_i (\mathbf{w} \cdot \mathbf{x}_i) \tag{3}$$

where $\mathcal{M}$ indexes the misclassifications.

Generalising to a Kernel Perceptron
However, this simplistic formulation struggles when it encounters a dataset that is not linearly separable. Hence, we introduce a kernel function to create a nonlinear separating surface. A kernel maps the data to a higher dimension, hence increasing the hypothesis space of the functions learned. The two kernel functions explored in this project are the polynomial kernel:

$$K_d(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d \tag{4}$$

where $d \in \mathbb{Z}, d > 0$

and the Gaussian kernel:

$$K_c(\mathbf{x}_i, \mathbf{x}_j) = e^{-c||\mathbf{x}_i - \mathbf{x}_j||^2} \tag{5}$$

Reformulating equation (2) in dual notation for a test example $\mathbf{x}_t$ we predict via:

$$\hat{y}_t = \text{sign}\Big(\sum_{i=1}^{m} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)\Big) \tag{6}$$

for a learned set of values $\boldsymbol{\alpha} \in \mathbb{R}^m$.

It is important to notice that the kernel perceptron necessarily retains all of the training data, $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)$ and each datapoint's corresponding alpha value, $\alpha_1, ..., \alpha_m$, in order to generate predictions for each new test example, x_t, which significantly increases the memory complexity when we compare this method to the vanilla perceptron.

Generalising to a multi-class Kernel Perceptron: "One-versus-All" Scheme
To generalise the above formulation to multi-class classification problem it is necessary to train an ensemble of binary perceptrons. We will first do this using the "One-versus-All" scheme.

In this scheme, J binary perceptrons are trained separately, where J is the total number of classes. For the Handwritten Digits dataset that we are using, $J = 10$. Each binary classifier, C_j, treats the j^{th} class of digits as the positive class ($y = +1$) and all other classes as the negative class ($y = -1$). Hence, the data which C_j is trained on is can be represented as:

$$D_j = \{(\mathbf{x}_i, y_{ij})\}_{i=1}^m \ : \ \mathbf{x}_i \in \mathbb{R}^n, \ y_{ij} = \begin{cases} -1; & class(\mathbf{x}_i) \neq j \\ +1; & class(\mathbf{x}_i) = j \end{cases}$$

The output of the j^{th} classifier for some test observation $\mathbf{x}_t$ can be thought of as the 'confidence' that $\mathbf{x}_t$ belongs to class j,

$$C_j(\mathbf{x}_t) = \sum_{i=1}^{m} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t) \tag{7}$$

The classification for observation $\mathbf{x}_t$ is performed by computing the values $C_j(\mathbf{x}_t), j = \{0, 1, .., 9\}$ and assigning $\mathbf{x}_t$ to the class which maximises the confidence:

$$\hat{y}_t = \underset{j}{\text{argmax}}(C_j(\mathbf{x}_t)) \tag{8}$$

Implementation of the "One-versus-All" Scheme
Using the above-outlined mathematical intuition, we implemented the '1-v-all' perceptron in Python. This section will explore the implementation decisions taken and their justifications.

Algorithm 1 One-versus-All Kernel Perceptron Algorithm

Input: Training Set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\},\ \mathbf{x}_i \in \mathbb{R}^{m \times n},\ y_i \in \{0, 1, \ldots, 9\}$

procedure KERNELPERCEPTRON

Initialise:

$\boldsymbol{\alpha} = \mathbf{0}^{m \times J}$

if *kernel* = *'polynomial'* **then**

$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d$

if *kernel* = *'gaussian'* **then**

$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-c||\mathbf{x}_i - \mathbf{x}_j||^2}$

Training:

for $epoch = 1, 2, \ldots, 5$ **do**

Initialise error to 0 at the beginning of each epoch

= 0

for *each observation* $(\mathbf{x}_t, y_t), t = \{1, \ldots, m\}$ **do**

Compute confidences for $\mathbf{x}_t$ belonging to each class

for *every class,* $j = \{0, \ldots, 9\}$ **do**

if $epoch = 1$ **then**

$C_j(\mathbf{x}_t) = \sum_{i=1}^{t-1} \alpha_{i,j} K(\mathbf{x}_i, \mathbf{x}_t)$

if $epoch \geq 2$ **then**

$C_j(\mathbf{x}_t) = \sum_{i=1}^{m} \alpha_{i,j} K(\mathbf{x}_i, \mathbf{x}_t)$

for end

Treat class j as the positive class

$$y_t^j = \begin{cases} -1 \text{ if } y_t \neq j \\ +1 \text{ if } y_t = j \end{cases}$$

Perform the update

if $C_j(\mathbf{x}_t) \cdot y_t^j \leq 0$ **then**

$\alpha_{t,j}^{new} = \alpha_{t,j}^{old} + y_t^j$

Perform classification

$\hat{y}_t = \underset{j}{\operatorname{argmax}}(C_j(\mathbf{x}_t))$

Check if classification correct

if $\hat{y}_t \neq y_t$ **then**

$errors = errors + 1$

for end

for end

Return the error for the class epoch

return: $\frac{errors}{m} \times 100$

To build a model that generalises well it is important to test its performance on unseen data. As the data available is limited, it is necessary to create numerous instances of the perceptron, each time randomly assigning the data to the train set or to the test set. To facilitate easily making new instances of the perceptron and accessing some of the internal variables a `KernelPerceptron` class is created.

During training, we pass through all the training datapoints numerous times - this is shown in the pseudocode by the *epochs*. Through experimentation we find that the algorithm consistently converges after passing through the data 5 times, hence this is performed for all computations that follow.

Computational complexity has been improved through pre-computing the training and test kernel matrices when the class is initiated for all observations using matrix multiplication, rather than computing individual dot products for each observation. The polynomial kernel matrix is computed by simply taking pairwise dot products of all the observations, storing them in a matrix and raising each element to the power of d. The Gaussian Kernel matrix is computed through first obtaining all the pairwise Euclidean Distances of all the observations and storing them in a matrix. Next, we square the matrix, multiply it by the factor of $-c$ and take the exponential.

A key aspect of online learning is the sequential updating of the weights vector, $\mathbf{w}$ (in primal notation) or alpha vector (in dual notation) for every new observation x_t. Hence, it initially seems necessary to create a loop which iterates through the entire dataset. However, it is possible to vectorize and compute the 'confidences', $C_j(\mathbf{x_t}) \; \forall \; j$ in a single matrix multiplication for a given $\mathbf{x_t}$, which significantly accelerates the computation. To facilitate this, the individual $\alpha_{i,j}$ values are stored in a $m \times J$ matrix, where m is the number of training examples and J is the number of classes, hence 10 for the digits dataset. Similarly, the kernel matrices used for the training and test set are stored in matrices which are $m \times m$ and $m \times r$ respectively, where r is the number of test observations.

In order to simultaneously evaluate the 'confidence' of $\boldsymbol{x}_t$ for each class, $C_j(\mathbf{x}_t) = \sum_{i=1}^{m} \alpha_{i,j} K(\mathbf{x}_i, \mathbf{x}_t)$, we can multiply the $m \times J$ matrix $\boldsymbol{\alpha}$ by the t^{th} column of the kernel matrix, $\mathbf{K}_t$, returning a J-dim vector $C(\mathbf{x}_t) = \boldsymbol{\alpha}\mathbf{K}_t$ of 'confidence' values. These are then stored in a global 'confidences' matrix. It is worth noting that when we compute the above summations during Epoch 1, the index on the sum is $t-1$, while for all subsequent epochs the index changes to m. This is because we initialise all α-values to be 0. Hence when computing the confidence values for the t^{th} point in the first epoch, only the points $i = \{1, \ldots, t-1\}$ may have been updated to have non-zero values. For all subsequent epochs, terms for all m training samples are included in the sum, as by this point all values of alpha have been updated.

The classification is performed through assigning $\mathbf{x_t}$ to the class which maximises the 'confidence', by finding the index of the maximum observation in the 'confidences' vector.

To ensure that the magnitude of the error is independent of dataset size and hence allow comparisons between eg. train and test errors, the error for each epoch is divided by m, the number of observations considered. For a single run, where 5 epochs are executed, error is initialised to zero at the beginning of each epoch and the error for the final epoch is reported

The test procedure is carried in an identical manner, except for the update step, which is omitted. The test kernel matrix is computed pairwise for all the training and test observations, hence it is no longer symmetric but rather $\mathbf{K} \in \mathcal{R}^{n \times m}$ where m is the size of the test set.

d	Mean % Train Error	Mean % Test Error
1	7.860984 $\pm$ 0.208242	9.217742 $\pm$ 1.275732
2	0.930358 $\pm$ 0.148839	3.811828 $\pm$ 0.473881
3	0.276956 $\pm$ 0.064253	3.051075 $\pm$ 0.352089
4	0.122345 $\pm$ 0.054263	2.836022 $\pm$ 0.361007
5	0.083356 $\pm$ 0.033638	2.698925 $\pm$ 0.444842
6	0.051761 $\pm$ 0.021374	2.741935 $\pm$ 0.399806
7	0.051089 $\pm$ 0.020654	2.795699 $\pm$ 0.310714

Table 1: Mean train and test errors for the '1-v-all' the Polynomial Kernel Perceptron, where d corresponds to the degree of the polynomial kernel. ($mean \pm std$)

1.1.1 Basic Results

In this section we implement a Polynomial Kernel Perceptron and investigate the effect of changing the degree of the polynomial on the train error and the test error. We investigate the effect of setting $d = 1, \ldots, 7$. For each value of d we train the Kernel Perceptron 20 times, each time on a different train-test split, keeping the train:test ratio at 80:20.

As can be seen from the error values (Table 1), by increasing the degree of the polynomial kernel we increase the flexibility of the decision surface and hence the training error of our model decreases as d is increased. A similar trend is spotted in the test error, until we get to $d = 6$, for which the test error begins to increase. This is likely indicative that the polynomial perceptron of degree 6 or 7 begins to over-fit our data. To find out which d-value allows the model to generalise best to the data, we will apply the technique of cross-validation.

1.1.2 Cross-validation

We investigate which value of d minimizes the average generalisation error, optimising the performance of the model on previously unseen data. To approximate the best selection of d, we used *5-fold cross-validation* as follows: we first split the data into training and test set, setting the ratio to 80:20. We then split the full 80% Training Set into 5 subsets. For each $d = 1, \ldots, 7$, we train a polynomial kernel of order d using 4 of the 5 subsets, then use this model to predict on the 5th subset, returning a cross-validation error for each setting of d. This process is repeated so that all combinations of 4/5 subsets are used as training sets, then the cross-validation errors for each d are averaged over. We select the setting of d with the lowest average cross-validation error to be the 'best' d^*. The polynomial kernel perceptron algorithm is then retrained on the full 80% Training Set using this d^*, then used to predict on the 20% Test Set in order to obtain a % test error.
We perform 20 runs of this procedure. For each run, the data is shuffled so that the 80:20 split is randomised, but shuffling is not implemented in cross-validation, to ensure all values of d are trained on the same data to allow comparison. We report the best d-value found for each run, and the performance on the 20% Test Set using this d^* in Table 2.

5-Fold Cross-Validation was implemented in the following manner, where the 'KFold' function from the sklearn library was used in step (1):

Run	**% Test Error**	d^*
1	2.258064516129032	6
2	2.741935483870968	7
3	2.849462365591398	6
4	2.6881720430107525	5
5	3.494623655913978	5
6	2.903225806451613	4
7	2.5806451612903225	6
8	3.010752688172043	5
9	2.6344086021505375	5
10	2.4193548387096775	4
11	3.1720430107526885	6
12	2.258064516129032	4
13	2.903225806451613	5
14	2.5268817204301075	5
15	2.6881720430107525	5
16	2.6344086021505375	7
17	3.3333333333333335	4
18	2.5806451612903225	7
19	2.6881720430107525	4
20	2.6344086021505375	4

Table 2: Percentage Test Error for the the '1-v-all' Polynomial Kernel of order d^*, selected using cross-validation.

K-Fold Cross-Validation

---

1. Split the data into K parts of roughly equal-size: $(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{(K)}, \mathbf{Y}^{(K)})$. Save the indices for each section.
2. **for** $k = 1, 2, \ldots, K$ **do**:
3. Train on the $K-1$ parts $(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})$ where $i \neq k$ and test on $(\mathbf{X}^{(k)}, \mathbf{Y}^{(k)})$. Compute cross-validation error, $error_k$.
4. **end for**
5. **return :** Average Cross-Validation Error: $\sum_{k=1}^{K} \frac{error_k}{K}$

'One-versus-All' Polynomial Kernel
Mean % Test Error: 2.8 ± 0.3
Mean *d:** 5 ± 1

We find that the model consistently generalises best when using $d = 5 \pm 1$. Now that we have completed model selection, we will investigate the capabilities of our model and attempt to understand its misclassifications.

1.1.3 Confusion Matrix

Firstly, we are interested in finding which pairs of digits are most frequently misclassified. To achieve this, we construct a confusion matrix where the column index corresponds to the true class and the row index corresponds to the predicted class. The diagonal entries are 0s as we only record the misclassifications. As we wish to investigate the

performance of the fully trained model, we compute the confusion matrix based on the errors made by the model on the test data.

We implement the same scheme as above - we use 5-fold cross-validation to find best d^* for each of the 20 runs, then retraining on the full 80% Training Set. A confusion matrix is then computed for the 20% Test Set for each of the 20 runs. The average confusion matrix and standard deviation associated with each pair are shown in Figure 1. To facilitate analysis we also show the confusion matrix in the form of a heatmap in Figure 2, where the standard deviation values have been omitted.

	0	1	2	3	4	5	6	7	8	9
0	0.0 +/- 0.0	0.05 +/- 0.2179	0.4 +/- 0.8	0.8 +/- 0.9798	0.4 +/- 0.5831	0.55 +/- 0.669	0.75 +/- 0.9421	0.0 +/- 0.0	0.5 +/- 0.8062	0.25 +/- 0.433
1	0.0 +/- 0.0	0.0 +/- 0.0	0.2 +/- 0.4	0.15 +/- 0.3571	1.05 +/- 2.1558	0.0 +/- 0.0	0.45 +/- 0.5895	0.1 +/- 0.3	0.2 +/- 0.4	0.1 +/- 0.3
2	0.6 +/- 0.8602	0.15 +/- 0.3571	0.0 +/- 0.0	1.2 +/- 1.077	1.25 +/- 1.299	0.15 +/- 0.3571	0.35 +/- 0.7921	0.75 +/- 1.1347	1.1 +/- 1.2207	0.2 +/- 0.5099
3	0.3 +/- 0.4583	0.1 +/- 0.3	1.4 +/- 1.1136	0.0 +/- 0.0	0.0 +/- 0.0	2.1 +/- 1.179	0.0 +/- 0.0	0.35 +/- 0.5723	2.1 +/- 0.8888	0.05 +/- 0.2179
4	0.25 +/- 0.433	1.05 +/- 0.669	1.5 +/- 1.1619	0.0 +/- 0.0	0.0 +/- 0.0	0.45 +/- 0.5895	1.65 +/- 1.3519	0.7 +/- 0.8426	0.35 +/- 0.7921	1.8 +/- 1.99
5	1.1 +/- 1.0909	0.05 +/- 0.2179	0.6 +/- 0.6633	1.8 +/- 1.4	0.6 +/- 0.6633	0.0 +/- 0.0	1.65 +/- 1.5256	0.15 +/- 0.3571	0.7 +/- 0.9539	0.55 +/- 0.8047
6	0.95 +/- 0.669	0.4 +/- 0.4899	0.35 +/- 0.477	0.05 +/- 0.2179	1.05 +/- 1.3955	0.5 +/- 0.6708	0.0 +/- 0.0	0.0 +/- 0.0	0.5 +/- 0.5916	0.05 +/- 0.2179
7	0.0 +/- 0.0	0.2 +/- 0.5099	0.95 +/- 0.9734	0.35 +/- 0.5723	0.85 +/- 1.1079	0.2 +/- 0.4	0.0 +/- 0.0	0.0 +/- 0.0	1.05 +/- 1.3219	2.0 +/- 1.5492
8	1.45 +/- 1.0235	0.35 +/- 0.6538	1.25 +/- 0.9421	2.25 +/- 1.7854	0.45 +/- 0.7399	1.8 +/- 1.1662	0.5 +/- 0.6708	0.55 +/- 0.8047	0.0 +/- 0.0	0.55 +/- 0.669
9	0.4 +/- 0.4899	0.15 +/- 0.477	0.2 +/- 0.5099	0.3 +/- 0.6403	1.45 +/- 1.1169	0.15 +/- 0.3571	0.05 +/- 0.2179	1.8 +/- 1.4353	0.35 +/- 0.5723	0.0 +/- 0.0

Figure 1: Average confusion matrix, where each entry corresponds to average number of confusions for the given pair on a single run and the associated standard deviation. The row labels correspond to true class labels and the column labels to the predicted classes.

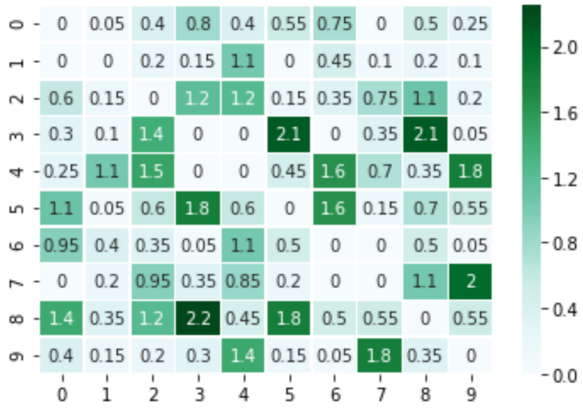

Figure 2: Average heatmap confusion matrix, where each entry corresponds to average number of confusions for the given pair on a single run. The row labels correspond to true class labels and the column labels to the predicted classes.

The confusion matrices show that the 5 most difficult to predict pairs are (8,3), (3,5), (3,8), (5,3) and (4,9) where in each pair the first digit corresponds to the true class and the second digit to the predicted class. All digits in the first 4 pairs have a rounded bottom half and a curved top half which is probably what makes them difficult to distinguish. Meanwhile the numbers 4 and 9 have a closed loop shape in their top half, and often both have a vertical line in their bottom half when written, which may make them appear similar.

1.1.4 Hardest Items to Predict

While certain pairs are particularly difficult to predict, some individual data items are difficult to predict not necessarily due to their true class but due to some unusual feature of the given observation. To obtain the 5 data items which are most difficult to predict we use the same 5-fold cross-validation procedure as used above for the confusion matrix, where for each run our final model is retrained on the full 80% Training Set using best d^* and tested on the 20% Test Set. Then we save the indexes of all misclassified observations from both the test set and the train set. The justification for this time considering mistakes made on both the training and test set arise from the fact that if the model fails to predict correctly an example that it has already seen then it must be a 'difficult' example - and this is precisely the type of datapoint which we are interested in. Once we complete all the runs, we total the number of times that en error was made on each of the m observations, then find the 5 examples which are most frequently misclassified. These are shown in Figure 3.

Analysing the images, it is obvious that the first 4 digits would be challenging to classify even for a human being as they are very disfigured. The first three 4s resemble straight vertical lines, so they are likely misclassified as 1s. The 7 is simply a horizontal line through the center of the cell. This does not have the defining features of a 7 that the algorithm is likely to have learned - a horizontal line in the top half, and a roughly verticle line descending from it. Lastly, the final 4 looks quite bent and somewhat rounded at the top, hence would likely be misclassified as a 9 by humans and perceptrons alike.

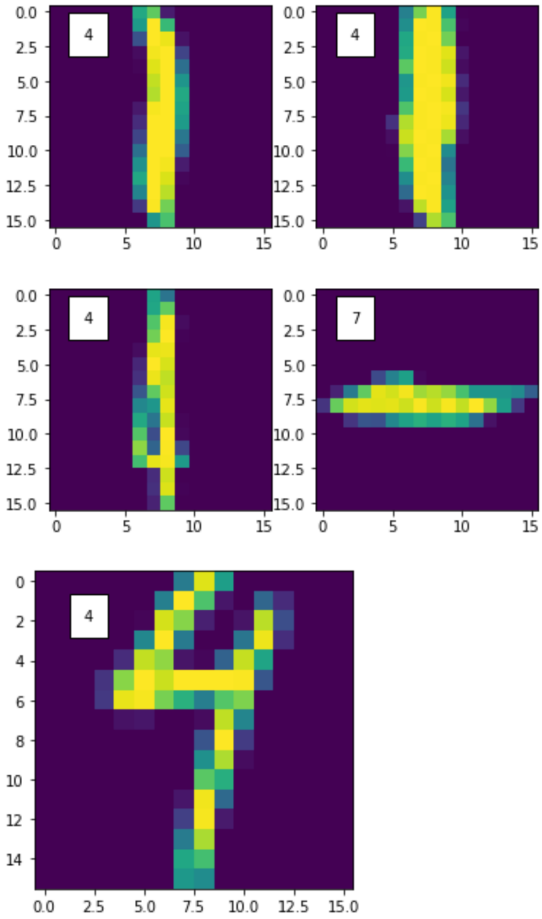

Figure 3: Visualisations of the 5 data items which are most difficult to classify correctly. The true class labels are shown in the top left corner of each image.

1.1.5 Implementing a Gaussian Kernel

Thus far we have only considered the polynomial kernel function, but the use of the dual-form makes it easy to apply any valid kernel function to the perceptron. We investigate the use of the Gaussian Kernel:

$$K_c(\mathbf{x}_i, \mathbf{x}_j) = e^{-c||\mathbf{x}_i - \mathbf{x}_j||^2} \tag{9}$$

The Gaussian kernel has many desirable properties, primarily because the functions space corresponding to the Gaussian kernel, $\mathcal{H}$ is very smooth, hence the resulting in a smooth function learned by the perceptron. This is because, given some arbitrary, bounded and continuous function h, there exists some function $f \in \mathcal{H}$ which lies close to g, where similarity is measured using $||\cdot||_\infty$. As this property is not shared by the polynomial kernel, we may expect the Gaussian Kernel to better generalise and hence result in lower test errors.

When the distance between x_i and x_j is very large in comparison with the coefficient c, the output tends to 0. Hence, the coefficient c scales the distances between the points and controls the type of classifier created. As the value of c tends to 0, the classifier becomes increasingly localised, while for larger c values a more general classifier is created.

c	Mean % Train Error	Mean % Test Error
0.008	0.156628 ± 0.035893	2.930108 ± 0.409274
0.009	0.120328 ± 0.031090	3.077957 ± 0.407992
0.010	0.098817 ± 0.036144	2.680108 ± 0.332845
0.011	0.089406 ± 0.029232	2.658602 ± 0.389729
0.012	0.063861 ± 0.026508	2.758065 ± 0.348468
0.013	0.057811 ± 0.024460	2.599462 ± 0.373366
0.014	0.055122 ± 0.020785	2.682796 ± 0.442660
0.015	0.051089 ± 0.019296	2.559140 ± 0.437239
0.016	0.041006 ± 0.023082	2.599462 ± 0.302357
0.017	0.042350 ± 0.018666	2.545699 ± 0.250029
0.018	0.050417 ± 0.023624	2.537634 ± 0.345762
0.019	0.034956 ± 0.015563	2.653226 ± 0.486633
0.020	0.036300 ± 0.016521	2.591398 ± 0.345762
0.021	0.038317 ± 0.021374	2.559140 ± 0.316885
0.022	0.034956 ± 0.018237	2.615591 ± 0.378746

Table 3: Mean train and test errors for the '1-v-all' Gaussian Kernel Perceptron, where c corresponds to the value of the coefficient in the Gaussian Kernel.

In the following section we will aim to find a suitable range, S, for the parameter c and perform cross-validation to finding optimum value of c. The computation of the Gaussian Kernel has been discussed in some detail in section 1.1, hence it is omitted here.

1.1.5.1 Basic results

In order to decide a reasonable set of values S to cross-validate c over, we ran some initial experimentation. We first considered the range of values $\{0.0001, 0.001, 0.01, 0.1, 1, 10, 100\}$. The best value of c in this set was $c = 0.01$. After more similar experiments, we decided to consider a range of numbers about the value 0.015. We consider the range $S = \{0.008, 0.009, 0.010, \ldots, 0.022\}$. A scheme identical to that of 1.1.1 is implemented, where for each value of c we train the Kernel Perceptron 20 times, each time on a different train-test split, keeping the split ratio 80:20. The results are reported in Table 3. We observe the lowest test errors for $c = 0.018$ which is a relatively small value. From this we can infer that a local classifier is more effective for the handwritten dataset than a more general one.

1.1.5.2 Cross-Validation

Using the above-defined S range we perform cross-validation to find c^*. The scheme implemented is identical to that in section 1.1.2. The results are reported in Table 4.

'One-versus-All' Gaussian Kernel
Mean % test error: 2.5 ± 0.3
Mean c^*: 0.018 ± 0.003

1.1.6 Discussion: Comparison of Polynomial and Gaussian Kernels

Somewhat surprisingly we find that when using the optimum parameters for both the Polynomial and Gaussian kernel perceptron there is no significant difference between the the mean test errors of both kernels, since the values lie within the bounds of uncertainty, as seen in Figure 4. We can only hypothesize as to why we both of kernels seem to perform equally well for this particular dataset.

Run	% Test Error	Best c
1	2.4193548387096775	0.016
2	3.1720430107526885	0.014
3	3.010752688172043	0.022
4	2.6881720430107525	0.021
5	2.795698924731183	0.02
6	2.3655913978494625	0.019
7	2.741935483870968	0.018
8	2.795698924731183	0.021
9	2.849462365591398	0.016
10	2.1505376344086025	0.015
11	2.6344086021505375	0.016
12	2.903225806451613	0.019
13	1.935483870967742	0.018
14	1.935483870967742	0.017
15	2.3655913978494625	0.014
16	2.311827956989247	0.021
17	2.4193548387096775	0.017
18	2.043010752688172	0.015
19	2.6344086021505375	0.014
20	2.741935483870968	0.02

Table 4: Percentage Test Error for the '1-v-all' Gaussian Kernel built using best c obtained using cross-validation.

1.2 One-versus-One Kernel Perceptron

Generalising to a multi-class Kernel Perceptron: "One-versus-One" Scheme
In this section we will explore an alternative method of generalising the binary perceptron for multi-class classification.

In the "One-versus-One" scheme N binary classifiers are built, where each classifier is trained on only 2 classes, treating one as the positive class and the other as the negative class. $N = \frac{C(C-1)}{2}$, where C is the number of classes. Thus 45 binary classifiers are required for the digit dataset. Each of the 10 digit classes is considered by 9 classifiers, against each of the remaining classes.

While some implementation details will be identical to that of a "One-versus-All" scheme, some steps need to be modified. Firstly, in '1-v-all' every classifier is trained on the entire dataset, while in the '1-v-1' every pair-classifier $C_{h,j}$ is only trained on the data belonging to classes h or j.

$$D_{h,j} = \left\{ (\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathcal{R}^d,\ y_i = \begin{cases} -1; & class(\mathbf{x}_i) = j \\ +1; & class(\mathbf{x}_i) = h \end{cases} \right\}_{i=1}^{n}$$

The output of every classifier $C_j(\mathbf{x_t})$ in '1-v-all' corresponded to a 'confidence' of $\mathbf{x}_t$ belonging to class j and the classification was made through assigning the digit to the class which maximised the confidence. In the '1-v-all' scheme, however, the output of every individual classifier $C_{h,j}(\mathbf{x_t})$ is an assignment of $\mathbf{x}_t$ to either class i or j.

$$C_{h,j}(\mathbf{x}_t) = \text{sign}(\sum_{i=1}^{m} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t) \tag{10}$$

Hence, classification is now performed through assigning $\mathbf{x}_t$ to the class most voted amongst the 45 classifiers.

$$\hat{y}_t = \text{mode}(\mathbf{C}) \tag{11}$$

where $\mathbf{C} \in \mathcal{R}^{45}$ and each entry of the vector corresponds to a prediction generated by one of the pair-classifiers.

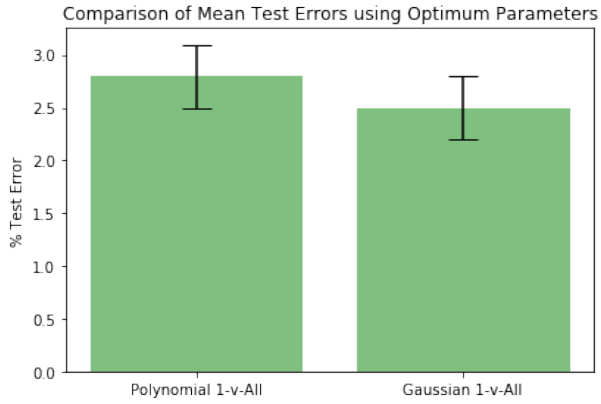

Figure 4: Comparison of Average Test Errors for the Polynomial and Gaussian kernels using the "One-versus-All" Scheme.

Implementation of the 'One-versus-One' Scheme

When considering how to best implement this scheme numerous implementations were explored and assessed. The key considerations when selecting and implementation were those of time complexity and retaining the sequential update steps, which are fundamental for online algorithms.

Intuitively, the most obvious way to approach this problem is to create 45 different datasets, one for each pair-classifier:

$$\{D_{h,j}^{(n)} = \{\mathbf{x}_i, y_i\}_{i=1}^{m}\}_{n=1}^{45}$$

where $y_i = h \vee j, \forall i$.

From this it naturally follows to create 45 kernel matrices and, since the size of each training set will vary, 45 alpha vectors for each classifier. However, as we want to be iterating through the observations individually and the dimensions of all the alpha and kernel matrices differ, for every observation we will have to perform 45 matrix multiplications during the training and likewise when computing predictions for a test set. Upon implementing this version of the algorithm we found it was very slow. Hence, we experimented with looping through every classifier, $C_{h,j}$ rather than through every observation and computing the assignments for all $\mathbf{x}_i$ in a single step for a given classifier through vectorizing. This implementation only required performing 10 matrix multiplications per epoch, hence the time complexity for each epoch decreased. We found, however, that this implementation took significantly longer to converge. We hypothesize this is because this implementation contradicts a key principle of online learning - sequential updates. Consider some data point $\mathbf{x}_t$: in computing the prediction for this data point all the preceding alpha values $\alpha_1, \ldots, \alpha_{t-1}$ influence the prediction. If all the alpha values are updated in a single step, rather than sequentially, it will cause fluctuations in the predictions and this explains increased convergence time.

Thus, another implementation was developed which allowed for performing sequential updates and for vectorizing the prediction step for a single $\mathbf{x}_i$ across all 45 classifiers. This was achieved through pre-computing a single training kernel, a single test kernel matrix and a single alpha matrix when initiating a Kernel Perceptron, as opposed to computing 45 instances of each matrix. The prediction step is performed in the same way as for the 'One-versus-All' perceptron, computing matrix multiplication of the alpha matrix with the k^{th} column of the kernel matrix.

We compute the predictions for a single observation $\mathbf{x}_t$ for all 45 classes in a single step, by computing the matrix multiplication $\alpha^T \mathbf{K}_t$, $\alpha \in \mathcal{R}^{m \times 45}$ and $\mathbf{K}_t$ denotes the t^{th} column of the kernel matrix, containing the dot products of x_t with all the training examples. This may seem couter-intuitive, since only the observations belonging to classes h or j should be influencing the prediction for classifier $C_{h,j}$, whereas here we seem to be considering all the observa-

d	Mean % Train Error	Mean % Test Error
1	3.543291 $\pm$ 0.177674	6.451613 $\pm$ 0.737559
2	0.308551 $\pm$ 0.078218	3.706989 $\pm$ 0.484013
3	0.081339 $\pm$ 0.044076	3.336022 $\pm$ 0.388243
4	0.049072 $\pm$ 0.018666	3.282258 $\pm$ 0.492889
5	0.043695 $\pm$ 0.021204	3.336022 $\pm$ 0.418691
6	0.035628 $\pm$ 0.025957	3.413978 $\pm$ 0.398539
7	0.037645 $\pm$ 0.025367	3.440860 $\pm$ 0.327472

Table 5: Mean train and test errors for the Polynomial '1-v-1' Kernel Perceptron, where d corresponds to the degree of the polynomial kernel.

tions. However, this is justified when we consider how the update step is performed - this is a key difference between the 'One-vs-All' and 'One-vs-One' algorithms. Just as in 'One-vs-All' all alpha values are initialised to be 0 hence initially none of the observations contribute to the prediction. Then at the update step we want to train every classifier $C_{h,j}$ only on the data points belonging to either of its classes. Hence, for a training datapoint $(\mathbf{x}_t, y_t)$ where $y_t = j$ we obtain the indexes of the classifiers which consider the j^{th} class (as either the positive or the negative class) and update the corresponding alpha values only for those classifiers. Thus, for classifiers which do not consider the j^{th} class the alpha value corresponding to $\mathbf{x}_t$ will always remain 0 and thus $\mathbf{x}_t$ will not influence the prediction. Hence, we have shown that this method is equivalent to explicitly creating separate datasets for each classifier.

Some conventions are introduced to facilitate indexing the data. For a given binary classifier which is predicting for the classes eg. [4,6] the first digit is treated as the positive class and the second digit as the negative class. We also store an array `classifiers` $\in \mathcal{R}^{45\times 2}$ in which every row corresponds to one of the classifiers and stores the 2 classes which the given classifier considers. We also compute `digit_indexes` $\in \mathcal{R}^{10\times 9}$ in which every row corresponds to one of the classes and stores the indexes of the classifiers which train on this digit. Lastly, we pre-compute `digit_signs`, a list of 10 vectors, one for each digit class. Each of the vectors contains 9 entries, each one corresponding to one of the 9 classifiers which consider the given digit and each entry is either -1 or +1, indicating whether the given digit is treated as the positive or the negative class in the given classifier.

1.2.1 Basic Results

We once again implement a Polynomial Kernel Perceptron and investigate the effect of changing the degree of the polynomial on the train error and the test error. We investigate a range of $d = 1, \ldots, 7$. For each value of d we train the Kernel Perceptron 20 times, each time on a different train-test split, keeping the train:test ratio at 80:20. The results are shown in Table 5.

It is interesting to note that the '1-v-all' d = 1 gave very high test errors of around $7 - 9\%$ for both, the training and test set, while for the '1-v-1' the train and test errors associated with d = 1 appear to be significantly smaller, at around $3.5 - 6.5\%$.

The test error appears to decrease as the value of d is increased to d=4 and starts to slowly increase afterwards, suggesting signs of over-fitting.

1.2.2 Cross-Validation

We perform 20 runs of this procedure. For each run, the data is shuffled so that the 80:20 split is randomised, but shuffling is not implemented in cross-validation, to ensure all values of d are trained on the same data to allow comparison. We report the best d-value found for each run, and the performance on the 20% Test Set using this d^* in Table 2.

'One-versus-One' Polynomial Kernel
Mean % test error: 3.4 $\pm$ 0.7
Mean *d: 4 $\pm$ 1

Algorithm 2 One-versus-One Kernel Perceptron Algorithm

Input: Training Set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\},\ \mathbf{x}_i \in \mathcal{R}^{n \times d},\ y_i \in \{0, 1, \ldots, 9\}$

procedure KERNELPERCEPTRON

Initialise:

$\alpha = 0$

if *kernel* = *'polynomial'* **then**

$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d$

if *kernel* = *'gaussian'* **then**

$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-c||\mathbf{x}_i - \mathbf{x}_j||^2}$

Training:

for $epoch = 1, 2, \ldots, 5$ **do**

errors = 0

for *every observation* $(\mathbf{x}_t, y_t), t = \{1, \ldots, n\}$ **do**

Compute confidences for $\mathbf{x}_t$ belonging to each class

for *every classifier,* $k = \{1, 2, \ldots, 45\}$ **do**

if $epoch = 1$ **then**

$C^k(\mathbf{x}_t) = \text{sign}(\sum_{i=1}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$

if $epoch \geq 2$ **then**

$C^k(\mathbf{x}_t) = \text{sign}(\sum_{i=1}^{n} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$

for end

Consider only data points the belonging to class h or j:

$$y_t^{\{h,j\}} = \begin{cases} +1; & y_t = h \\ -1; & y_t = j \\ 0; & y_t \neq j \ \& \ y_t \neq h \end{cases}$$

Perform the update

if $C^k(\mathbf{x}_t) \cdot y_t^{\{h,j\}} < 0$ **then**

$\alpha_{t,k}^{new} = \alpha_{t,k}^{old} + y_t^{\{h,j\}}$

Perform classification

$\hat{y}_t = mode(C^1(\mathbf{x}_t), \ldots, C^{45}(\mathbf{x}_t))$

Check if classification correct

if $\hat{y}_t \neq y_t$ **then**

errors $=$ *errors* $+ 1$

for end

for end

return: $\frac{\textit{errors}}{n} \times 100$

Run	% Test Error	Best d
1	2.956989247311828	4
2	4.408602150537634	4
3	5.43010752688172	3
4	3.118279569892473	4
5	3.4408602150537635	4
6	3.2795698924731185	5
7	4.354838709677419	3
8	3.870967741935484	6
9	3.2795698924731185	3
10	2.849462365591398	3
11	2.903225806451613	4
12	3.225806451612903	5
13	3.118279569892473	5
14	3.225806451612903	5
15	3.978494623655914	3
16	3.4408602150537635	6
17	2.903225806451613	6
18	2.4731182795698925	3
19	3.387096774193549	3
20	3.225806451612903	4

Table 6: Percentage Test Error for the Polynomial Kernel for the '1-v-1' built using best d obtained through cross-validation.

Comparison of 1 vs All and 1 vs 1 Perceptrons

The first and most significant difference between the two models is that of the number of perceptrons trained: the '1-v-All' scheme only requires 10 perceptrons, while the '1-v-1' requires a total of 45 binary classifiers, hence the latter requires significantly greater computational and time expense.

A priori, we hypothesized that the '1-v-1' would generalise better than the '1-v-all'. In 1-vs-all, the dataset size of the class for which a given classifier is being trained is much smaller than the dataset of 'negative' class. For instance, supposing that the number of training examples for each of the digits is roughly the same, when training a classifier for digit 0 versus digits 1-9, the proportion of the 0 dataset and the not-0 dataset is roughly 1:9. This is not ideal as the model might develop a bias for not-0, due to this disproportionality. It can also result in one class achieving dominance simply do to an imbalance in the dataset. Meanwhile, when training each individual classifier in the 1-vs-1 perceptron the proportions of the data are likely to be closer to being even as only 2 classes are considered at a time. This should, in principle, make the 1-vs-1 more robust to imbalances in the dataset.

However, this is not what we observed: the performance on the test set by the '1-v-all' was better than that of the '1-v-1'. We also observe a larger standard deviation for the mean test error the polynomial kernel for the 'One-versus-All' scheme, suggesting greater variability in how the model predicts. We hypothesize that this may be due to the majority vote scheme - the downfalls of this scheme are discussed in greater detail in section 1.5 Comparison of Algorithms.

1.3 K-Nearest Neighbours

Mathematical Formulation of KNN

While the perceptron requires a generalisation procedure to enable multi-class classification, some methods are inherently multi-class. One of these methods is K-Nearest Neighbours (KNN). KNN is a simple yet surprisingly powerful non-parametric algorithm which predicts the classification of a new test input by comparing the distance in input

space between the unseen sample and all 'training' samples for which we know the true label. There is no training step required, but rather all observations must be retained in order to be used for prediction.

In a classification setting, to classify a test example $\mathbf{x}_t$ firstly the similarities between $\mathbf{x}_t$ and all the observations in the training set are computed. In this algorithm we have chosen to use the L^2 norm ie. Euclidean Distance as a measure of similarity.

The euclidean distance between 2 observations, $\mathbf{x}_t$ and $\mathbf{x}_i$ is defined as:

$$||\mathbf{x}_t, \mathbf{x}_i|| = d(\mathbf{x}_t, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{d}(x_{j,t} - x_{j,t})^2} \tag{12}$$

The distances between the test observation $\mathbf{x}_t$ and all the training examples are aligned in as decreasing order such that:

$$||\mathbf{x}_{(1)} - \mathbf{x}_t|| \leq ||\mathbf{x}_{(2)} - \mathbf{x}_t|| \leq \cdots \leq ||\mathbf{x}_{(N)} - \mathbf{x}_t|| \tag{13}$$

and the training observations are stored in a an ordered list, which can be expressed as:

$$(\mathbf{x}_{(1)}, y_{(1)}), \cdots, (\mathbf{x}_{(N)}, y_{(N)}) \tag{14}$$

From Equation 14 we select the first k observations, which correspond to the k nearest neighbours of $\mathbf{x}_t$. The test observation $\mathbf{x}_t$ is then assigned to a class based on the majority vote of it's k-neighbours. It is worth noting a special case of k=1, where $\mathbf{x}_t$ is simply assigned the class of its closest neighbour.

For efficiency the pairwise distances between all test points, $\{\mathbf{x}_{(i)}, y_{(i)}\}_{i=1}^{m}$ and all the training points, $\{\mathbf{x}_{(i)}, y_{(i)}\}_{i=1}^{n}$ have been computed using the `euclidean_distances` function which has been discussed in greater detail in section 1.1.

To verify that our implementation is correct we compare the predictions generated by our algorithm to those generated by `KNeighborsClassifier` from the sklearn library. We expected to find that our predictions are identical since KNN is completely deterministic, and that indeed was the case.

It is important to note that while later on we refer to 'training' the KNN, we are using this term loosing and we are simply referring to acquiring the training data since KNN does not require any learning to perform predictions.

1.3.1 Basic Results

In model selection we will aim to find an optimum value of k for which the model generalises well, where k corresponds to the number of neighbours considered when classifying the test point.
First we perform some heuristic experimentation to find a suitable range of k-values, experimenting with values ranging from $k = 1$ to $k = 20$. We find are tighten our range to $S = \{1, 2 \cdots, 9\}$ For each value of d we train the KNN 20 times, each time on a different train-test split, keeping the train:test ratio at 80:20. The results are shown in Table 7. We find the smallest test error occurs for k = 1, hence a very localised model works best. This is consistent with what we found using the Gaussian Kernel, where a small distance scaling coefficient was preferred.

1.3.2 Cross-Validation

To determine the k-value which leads to lowest generalisation error we perform cross-validation using the scheme described in the previous section.

KNN
Mean % test error: 3.3 $\pm$ 0.4
Mean *k: 1.3 $\pm$ 0.7

k	**Mean % Train Error**	**Mean % Test Error**
1	0.000000 ± 0.000000	3.198925 ± 0.478011
2	1.876176 ± 0.066339	4.223118 ± 0.389729
3	1.840549 ± 0.080768	3.534946 ± 0.396129
4	2.400511 ± 0.094826	3.696237 ± 0.367356
5	2.573272 ± 0.096987	3.728495 ± 0.431207
6	2.945012 ± 0.118317	4.126344 ± 0.486812
7	3.016268 ± 0.147906	4.309140 ± 0.470323
8	3.375908 ± 0.129298	4.295699 ± 0.452668

Table 7: Mean train and test errors for the KNN classifier, where k corresponds to the number of neighbours considered when assigning an observation to a class.

Run	**% Test Error**	**Best** k
1	3.1720430107526885	1
2	4.46236559139785	1
3	3.763440860215054	1
4	2.849462365591398	1
5	2.741935483870968	1
6	3.2795698924731185	1
7	3.1720430107526885	1
8	3.7096774193548385	1
9	3.064516129032258	1
10	3.3333333333333335	3
11	3.3333333333333335	3
12	3.7096774193548385	1
13	3.3333333333333335	1
14	3.6021505376344085	1
15	3.7096774193548385	1
16	2.903225806451613	1
17	3.655913978494624	1
18	2.849462365591398	3
19	2.5268817204301075	1
20	3.118279569892473	1

Table 8: Percentage Test Error for the KNN Classifier built using best k obtained through cross-validation.

1.4 Support Vector Machine (SVM)

Mathematical Formulation of the SVM

We consider, first, the problem of binary classification using the SVM algorithm[1]. Given a set of training data $(\boldsymbol{x}_i, y_i) \in \mathbb{R}^n \times \{-1, 1\}$, $i = 1, \ldots, m$, the soft-margin SVM seeks to solve the primal optimisation problem:

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^\intercal\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \quad i = 1, \ldots, m \\ & \xi_i \geq 0, \qquad\qquad\qquad\quad i = 1, \ldots, m \end{aligned} \tag{15}$$

where $\phi(\mathbf{x}_i)$ maps $\mathbf{x}_i$ to some feature space, and $C > 0$ is the regularisation parameter.

The use of a potentially high-dimensional feature map $\phi(\cdot)$ makes it favourable to rephrase this as a dual problem:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}Q\boldsymbol{\alpha} - \sum_{i=1}^{m}\alpha_i \\ \text{subject to} \quad & \boldsymbol{y}^\intercal\boldsymbol{\alpha} = 0 \\ & 0 \leq \alpha_i \leq C \qquad i = 1, \ldots, m \end{aligned} \tag{16}$$

where $Q_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$, $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\intercal\phi(\mathbf{x}_j)$ is the kernel function, and α_i are the *Lagrange multipliers*. Once the dual optimisation problem has been solved for $\boldsymbol{\alpha}^*$, the solution to the primal problem may be found via

$$\mathbf{w}^* = \sum_{i=1}^{m} y_i\alpha_i^*\phi(\mathbf{x}_i)$$

and predictions are made via

$$\hat{y}_t = sign(\mathbf{w}^{*\intercal}\phi(\mathbf{x}_t) + b) = sign\Big(\sum_{i=1}^{m} y_i\alpha_i^* K(\mathbf{x}_i, \mathbf{x}_t) + b\Big)$$

A solution to the dual optimisation problem is optimal if all values of the Lagrangian multipliers satisfy the *KKT Conditions*. For this problem, we may conclude that an optimal solution has been found if, for all $i = 1, \ldots, m$:

$$\begin{aligned} \alpha_i = 0 \quad & \Leftrightarrow y_i\hat{y}_i \geq 1 \\ 0 < \alpha_i < C \quad & \Leftrightarrow y_i\hat{y}_i = 1 \\ \alpha_i \geq C \quad & \Leftrightarrow y_i\hat{y}_i \leq 1 \end{aligned} \tag{17}$$

where $\hat{y}_i$ indicates the output of the SVM for the training sample $\mathbf{x}_i$.

[1] (https://www.csie.ntu.edu.tw/ cjlin/papers/libsvm.pdf

The SMO algorithm avoids the computational complexity of working with the $m \times m$ Q matrix that features in the dual optimisation problem. SMO iteratively optimises the objective function for only two Lagrange multipliers at a time. Any two Lagrange multipliers may be optimised simultaneously due to the linearity constraint imposed in (16). In order to select which two Lagrange multipliers it should optimise at any time step, SMO loops through the set of training data in order to find any choice of i for which α_i does not satisfy the KKT conditions. The second Lagrange multiplier is then chosen to maximise the size of the step taken by doing joint optimisation.

Having selected two values to optimise, α_1, α_2, SMO optimises the objective function as follows:
We first note that, due to the inequality constraints in (16), the space of possible choices of (α_1, α_2) are restricted to be inside a 2d square of length C. Furthermore, due to the linearity constraint, we note that the possible selections of both lie on a diagonal line in this space. We denote the second derivative of the objective function along this line to be

$$\eta = 2K_\gamma(\mathbf{x}_1, \mathbf{x}_2) - K_\gamma(\mathbf{x}_1, \mathbf{x}_1) - K_\gamma(\mathbf{x}_2, \mathbf{x}_2)$$

We also define the following constants to be bounds on the possible values of α_2:

$$\begin{cases} \text{If } y_1 \neq y_2, \text{ define } L = \max\{0, \alpha_2^{old} - \alpha_1^{old}\}, & H = \min\{C, C + \alpha_2^{old} - \alpha_1^{old}\} \\ \text{If } y_1 = y_2, \text{ define } L = \max\{0, \alpha_2^{old} + \alpha_1^{old} - C\}, & H = \min\{C, \alpha_1^{old} + \alpha_2^{old}\} \end{cases}$$

SMO then computes the minimum setting of $\alpha_2^{new} = \alpha_2^{old} - \frac{y_2(E_1 - E_2)}{\eta}$, where $E_i = \hat{y}_i - y_i$, the error on the ith training sample.

Then, we update

$$\alpha_2^{new,clipped} = \begin{cases} H; & \alpha_2^{new} \geq H \\ \alpha_2^{new}; & L < \alpha_2^{new} < H \\ L; & \alpha_2^{new} \leq L \end{cases}$$

and we trivially update α_1 as

$$\alpha_1^{new} = \alpha_1 + y_1 y_2 (\alpha_2 - \alpha_2^{new,clipped})$$

The threshold, b, is finally updated at the end of each step in order to ensure that both α_1 and α_2 satisfy the KKT conditions.
The SMO algorithm continues to perform these updates until all datapoints in the training set satisfy the KKT conditions up to some precision ϵ. At this point, the algorithm terminates.

Implementation of the SVM

We will be using the sklearn function `sklearn.svm.SVC` in order to implement our multi-class SVM. We will consider using either the Polynomial or Gaussian kernel, but in either case will need to optimise over both the kernel hyperparameter, and the regularisation parameter $C > 0$.
This implementation of the SVM performs multi-class classification via the "One-Vs-One" method described in section 1.2 above, such that each of the 1-vs-1 classifiers are set up as a binary classification SVM, described on the previous page. This implementation additionally uses the efficient Sequential Minimal Optimisation (SMO) algorithm[2].

1.4.1 Basic Results

According to the NFL theorem there are no guarantees for one kernel to work better than the other, hence we need to perform empirical experimentation to decide where to use the Polynomial Kernel or the Gaussian Kernel. Experimenting with a range of γ and c values on both kernels we find that the Gaussian Kernel performs better on the test set. As the values of γ and c influence each other, ideally one should perform cross-validation over a range of γ and c,

[2]The details of this algorithm are outlined further in the original paper by Platt, 1998: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf

c	% Test Error
0.0001	82.95698924731182
0.001	82.95698924731182
0.01	24.408602150537632
0.1	6.344086021505376
1	3.225806451612897
10	2.4731182795699027
100	2.4731182795699027

Table 9: Test Errors for various values of regularization parameter c

k	Mean % Train Error	Mean % Test Error
1	0.155956 ± 0.030183	2.497312 ± 0.269072
10	0.022183 ± 0.007694	2.209677 ± 0.241577
20	0.010756 ± 0.005378	2.376344 ± 0.356059
30	0.010083 ± 0.005822	2.276882 ± 0.368299
40	0.010756 ± 0.005378	2.370968 ± 0.304559
50	0.010756 ± 0.005378	2.311828 ± 0.327913
60	0.010756 ± 0.005378	2.395161 ± 0.292245
70	0.008739 ± 0.006413	2.241935 ± 0.286565
80	0.008739 ± 0.006413	2.486559 ± 0.260003

Table 10: Mean train and test errors for the SVM classifier, where c corresponds to the value of the regularization parameter, for $\gamma - 0.017$.

in a grid-search style. However, due to time constraints a series of linear searches was performed in the heuristic stage and this was used to decide the value of γ and the range S to perform cross-validation for over c.

Firstly, we will decide on a suitable γ value - we use our initial experimentation to guide our range and we experiment, keeping the value of $c = 10$ constant. We begin with a broader range of $\gamma = \{0.001, 0.005, 0.01, 0.025, 0.05\}$ and subsequently we tighten our range to $\{0.009, 0.01, \ldots, 0.017, 0.018\}$. and find our optimum $\gamma^* = 0.017$. Earlier, keeping in turn the value of γ constant we searched for a suitable range S for c, starting with a logarithmic scale to explore a wide range of values. Performing a single run for each of the c-values we obtained the results shown in Table 9.

Now, having our optimum value γ^* we narrow down range of S to $\{1, 10, 30, 50, 70, 100\}$ and perform 20 runs for each value of c. The results are shown in Table 10. It is interesting to notice that while the magnitude of the scale is much larger that in the previous cases, the differences in the magnitude of the error are only around 0.1%.

From these results, it is difficult to draw any concrete conclusions regarding optimum c, since we have no measure of uncertainty. From looking at the training error, it is clear that the model learns the training data increasingly well, but there are no obvious signs of over-fitting in the test error. To find $c*$ we will implement cross-validation and obtain a measure for uncertainty.

SVM
Mean % test error: 2.2 ± 0.3
Mean c*: 13 ± 10

To interpret these findings, we need to consider that c behaves as a regularization parameter. A smaller value of c encourages a larger margin, hence a simpler decision function, allowing for some mis-classifications in the training set. Meanwhile a larger c a smaller margin is encouraged, provided all training examples are classified correctly. A value of $c = 10$, which is what we find our c^* to be, is relatively small, suggesting that a simpler decision boundary is preferred. This may indicate why, in the case of the '1-v-All' perceptron, we find that the polynomial kernel performs as well as the Gaussian kernel: the simpler boundary generated using a polynomial kernel allows the model to generalise better for this particular dataset.

Run	% Test Error	Best c
1	2.0430107526881613	10
2	2.8494623655913927	10
3	1.6666666666666714	10
4	1.6666666666666714	10
5	2.0430107526881613	30
6	2.6344086021505433	10
7	2.204301075268816	10
8	2.4731182795699027	10
9	2.204301075268816	10
10	1.8817204301075208	10
11	2.4193548387096797	10
12	2.4731182795699027	10
13	2.311827956989248	10
14	1.8817204301075208	10
15	2.0967741935483843	10
16	2.204301075268816	10
17	1.827956989247312	10
18	2.258064516129039	10
19	1.827956989247312	10
20	2.5806451612903203	50

Table 11: Percentage Test Error for SVM built using best c obtained through cross-validation and $\gamma = 0.17$

1.5 Comparison of Algorithms

Throughout this section we compared a total of 4 different classification models: a kernelised perceptron, generalised using the "One-versus-All" scheme, a kernelised perceptron, generalised using the "One-versus-One" scheme, K-Nearest Neighbours and SVM.

First, we will explore the performance of each model in terms of its ability to generalise. In Figure 5 the Mean Test Error for each model has been plotted along with the associated standard deviation. The greatest significant difference is that of the performance of the SVM in comparison with the Polynomial "One-versus-One" perceptron and KNN, where SVM achieves the lowest test error of 2.2 ± 0.3 while the polynomial "One-versus-One" perceptron and KNN achieved $3.4 \pm 0.7\%$ amnd $3.3 \pm 0.4\%$ respectively. The RBF "One-versus-All" Perceptron also outperformed KNN, achieving the second lowest test error of 2.5 ± 0.3. The variances for polynomial "One-versus-One", RBF "One-versus-One" and SVM are all quite small at 0.3, while the standard deviation for the "One-versus-One" scheme is very large at 0.7. Such great variation in this model is could be caused by the voting scheme. For instance, if a digit is a '3' but resembles a '8' then all the 9 classifiers which consider 8s are likely to predict an 8, hence influencing the result of the voting and increase the variation.

While KNN's error is by no means the smallest, objectively an error of 3.3% achieves an accuracy of 96.7%, which is quite high, especially when one considers that no training is required. However, this method and the kernelised perceptrons require retaining all of the training data to produce a prediction, which can lead to long computational times for datasets significantly larger than the one considered here.

In terms of time complexity for this particular dataset our implementation of the '1-v-All' Perceptron was significantly faster than all other models, training and generating a prediction on the entire dataset in just 7 seconds. This is likely due to an efficient implementation, discussed in greater detail in section 1.1. Meanwhile the '1-v-1' implementation took significantly longer than all the other algorithms. This is likely due to the fact that the '1-v-1' requires creating , for this particular dataset, 45 different perceptrons.

	Polynomial 1-v-All	RBF 1-v-All	Polynomial 1-v-1	KNN	SVM
Mean % Test Errors	2.8 ± 0.3	2.5 ± 0.3	3.4 ± 0.7	3.3 ± 0.4	2.2 ± 0.3

Table 12: Comparison of Mean % Test Errors, where each model has been trained using optimum parameters. $(mean \pm std)$

Overall, considering both time complexity and ability to generalise, we find that the RBF '1-v-All' and the SVM outperform other models.

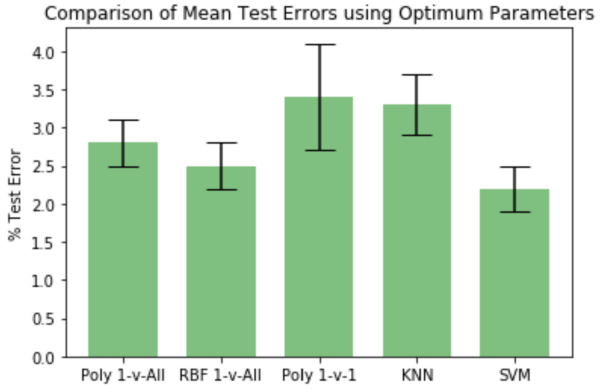

Figure 5: Comparison of Mean % Test Errors, where each model has been trained using optimum parameters.

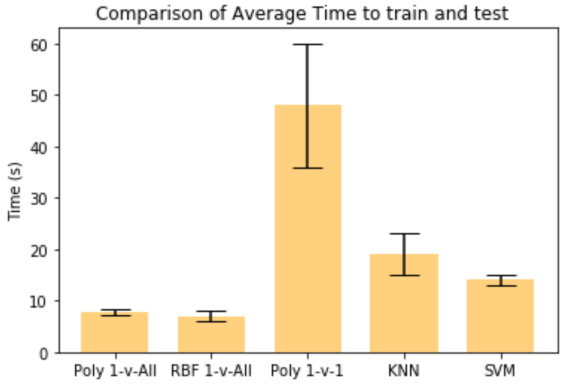

Figure 6: Comparison of time taken to train each model to convergence and generate a prediction on the test set, using the full dataset.

2 PART II

2.1

We present the plots of the estimated sample complexity for each of the four specified algorithms. The method for estimating the sample complexity of each algorithm for set n is discussed below. We note that the sample complexity for the 1-nearest-neighbour algorithm is very computationally expensive to find exactly, and even to estimate. Hence we have only estimated this value up to $n = 16$. All others have been estimated for up to $n = 100$.

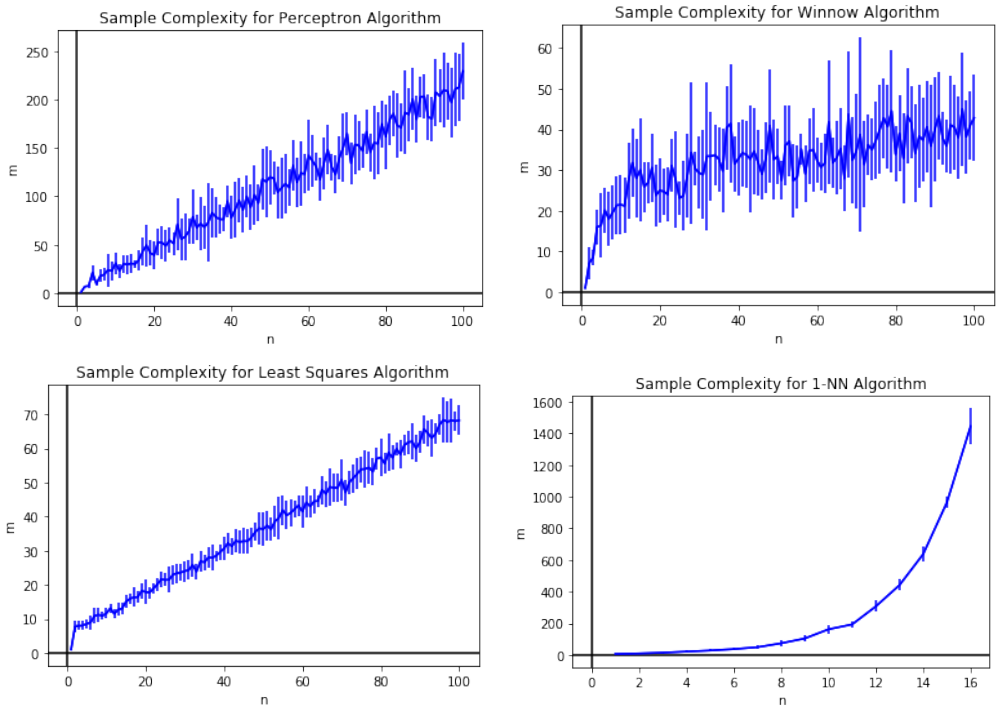

Figure 7: Plots of estimated sample complexity for Perceptron, Winnow, Least Squares and 1-NN algorithms. On each, we plot n, dimensionality of data, against m, estimated sample complexity, averaged over 10 trials. The vertical bars display one standard deviation above/below the mean.

2.2

As proposed, it is too computationally expensive to compute the sample complexity of these algorithms for large n exactly. We therefore must trade-off the accuracy of our computation with computation time. In this problem, we have defined the sample complexity to be the minimum value of m for which we achieve the bound $\mathbb{E}(\mathcal{E}(\mathcal{A}_{\mathcal{S}_m}) \leq 0.1$. In order to bound the expected generalisation error in this way, we consider the following theorem from the lectures:
For an algorithm $\mathcal{A}$ trained on some dataset $\mathcal{S}$, then for any $\delta \in (0, 1)$, with probability at least $1 - \delta$ over the random sample V of size r from $\mathcal{D}$ we have that

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) \leq L_V(\mathcal{A}_{\mathcal{S}}) + \sqrt{\frac{\ln \frac{1}{\delta}}{2r}}$$

where $L_V(\mathcal{A}_{\mathcal{S}})$ denotes the empirical error made on the sample V.

In this question, we seek to estimate the sample complexity, defined to be the smallest value of m for which

$$\mathbb{E}_{\mathcal{S}_m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq 0.1$$

For a specific sample of the training set $\mathcal{S}_m$, and a specific sample of the test set V, we can ensure with 90% certainty that the generalisation error of the algorithm is ≤ 0.1 using only the empirical error by ensuring that:

$$\begin{aligned}\mathcal{E}(\mathcal{A}_{\mathcal{S}}) \leq L_V(\mathcal{A}_{\mathcal{S}}) + \sqrt{\frac{\ln 10}{2r}} \leq 0.1 \\ \Rightarrow L_V(\mathcal{A}_{\mathcal{S}}) \leq 0.1 - \sqrt{\frac{\ln 10}{2r}}\end{aligned}$$

In our implementation, we have defined convergence to be when the algorithm has obtained a generalisation error lower than that specified by this bound.
For the Perceptron, Winnow and Least Squares algorithms, we have used a test size of 10000 samples. Therefore, the algorithm is considered to have converged only once it has demonstrated the ability to achieve an empirical error of

$$L_V(\mathcal{A}_{\mathcal{S}_m}) \leq 0.1 - \sqrt{\frac{\ln 10}{20000}} \approx 0.08927$$

whereas, for the 1-nearest neighbour algorithm, the computational complexity of the algorithm is much larger. Therefore we decide to test on a set of size 1000. Therefore the empirical error of the algorithm on a set of this size must instead reach

$$L_V(\mathcal{A}_{\mathcal{S}_m}) \leq 0.1 - \sqrt{\frac{\ln 10}{2000}} \approx 0.06607$$

We note three main problems with this approach. The first is that we are only able to obtain a probabilistic bound for the generalisation error. Therefore we cannot guarantee that the generalisation error is truly bounded by 0.1 by the time we consider the algorithm 'converged'. The second is that we are only bounding the generalisation error for a specific setting of $\mathcal{S}_m$, not the expectation of this value. In order to mitigate against this issue, we would ideally like to find the value of $\mathbb{E}[L_V(\mathcal{A}_{\mathcal{S}_m})]$, as

$$\mathbb{E}_{\mathcal{S}_m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}})] \leq \mathbb{E}_{\mathcal{S}_m}[L_V(\mathcal{A}_{\mathcal{S}})] + \sqrt{\frac{\ln 10}{2r}}$$

with 90% probability.
Although in our implementation, we do not approximate this value (which would involve testing on the same test set with a range of training sets).
Our implementation estimated the sample complexity of an algorithm by finding the lowest value of m for which $L_{V_r}(\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})$ was below the required value as calculated above for a specific sample of $(\mathcal{S}_m, V)$, then repeated this process 10 times. The sample complexities found were then averaged over, in an attempt to average over the setting of $(\mathcal{S}_m, V)$ and reduce the error introduced by using probabilistic bounds.
This calculation error will not have been removed completely, introducing bias into our estimations.

Another source of bias is that we would only consider a model to have *converged* if its empirical error was less than the required amount for the 5 most recent values of m. This obviously introduces large bias to the estimated sample complexity, $\hat{m}$, as the model is likely to have truly converged at $\hat{m} - 5$.

2.3

We consider the following asymptotic bounds on the sample complexity of each algorithm:

Perceptron: Based on the plot shown in Figure 8, we empirically propose the asymptotic bound $m(n) = \Theta(n)$.
Winnow: Based on the plot shown in Figure 9, we empirically propose the asymptotic bound $m(n) = \Theta(\log(n))$.
Least Squares: Based on the plot in Figure 10, we empirically propose the asymptotic bound $m(n) = \Theta(n)$.
1-NN: For the 1-nearest neighbour algorithm we were only able to estimate the sample complexity up to $n = 16$, therefore attempting to propose a tight asymptotic bound empirically does not seem logical. However, based on the plot shown in Figure 11, we propose an asymptotic bound lower bound of $m(n) = \Omega(n^2)$, and based on that shown in Figure 12, we additionally propose the asymptotic upper bound $m(n) = O(2^n)$.

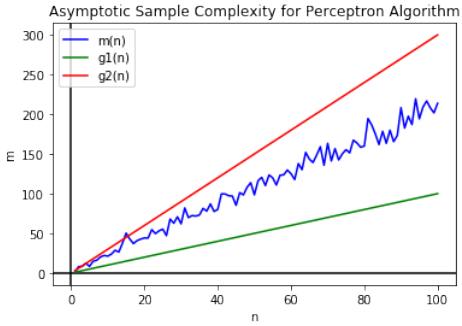

Figure 8: We plot the estimated sample complexity of the Perceptron as well as the functions $g_1(n) = n$, $g_2(n) = 3n$.

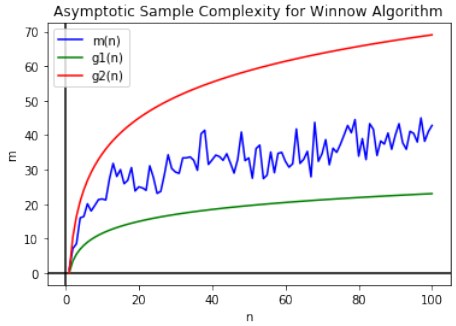

Figure 9: We plot the estimated sample complexity of the Winnow algorithm, as well as the functions $g_1(n) = 5\log(n)$, $g_2(n) = 15\log(n)$.

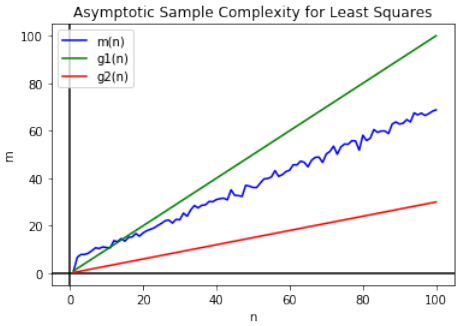

Figure 10: We plot the estimated sample complexity of the Winnow algorithm, as well as the functions $g_1(n) = \frac{1}{3}n$, $g_2(n) = n$.

As well as these empirically-founded asymptotic bounds, we also look to derive them analytically. In particular, we note that as we look to bound the generalisation error in order to find the sample complexity, we are effectively treating this as a PAC Learning problem. This means that we can use results from Learning Theory to bound the sample complexity $m_\lambda(\epsilon, \delta)$ based on the size and VC Dimension of the hypothesis space, $\mathcal{H}$, considered by an algorithm. We will use the following results from lectures:
Theorem 1: Let $\mathcal{H}$ be a finite hypothesis class $\Rightarrow \mathcal{H}$ is PAC learnable with sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leq \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$.
Theorem 2: 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

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

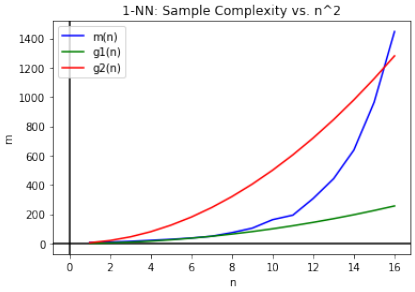

Figure 11: We plot the estimated sample complexity of the 1-NN algorithm, as well as the functions $g_1(n) = n^2$, $g_2(n) = 5n^2$.

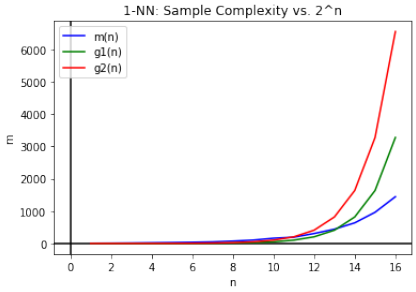

Figure 12: We plot the estimated sample complexity of the 1-NN algorithm, as well as the functions $g_1(n) = \frac{2^n}{20}$, $g_2(n) = \frac{2^n}{10}$.

Theorem 3: VCdim for Halfspaces:
For $\mathcal{X} = \mathbb{R}^n, \mathcal{H} = \{\mathbf{x} \to sign(\langle \mathbf{w}, \mathbf{x} \rangle) : \mathbf{w} \in \mathbb{R}^n\}$:

- Any arbitrary set $C = \{\eta_1, \ldots, \eta_n\} \subset \mathcal{X}$ is shattered by $\mathcal{H}$.

- Any set $C \subset \mathcal{X}$ such that $|C| = n + 1$ cannot be shattered.

We recognise first and foremost that the hypothesis class outlined in Theorem 3 is exactly that used by the Perceptron and Least Squares algorithms for binary classification. Therefore, for both of these algorithms we have that VCdim($\mathcal{H}$) $= n$. Therefore, we observe analytically (using Theorem 2) that the sample complexity of these algorithms is bounded, for some constants C_1, C_2, by

$$10C_1 \cdot VCdim(\mathcal{H}) + \log(1/\delta) \leq m_{\mathcal{H}}(0.1, \delta) \leq 10C_2 \cdot VCdim(\mathcal{H}) \log(1/\epsilon) + \log(1/\delta)$$

Thus, we have found analytically that $m(n) = \Theta(n)$ for both the Perceptron and Least Squares algorithms.

We also, more broadly, consider the space of all possible hypotheses for the 'just a little bit' problem. Due to the way the problem is formulated, this is a finite hypothesis space $\mathcal{H}$ such that $|\mathcal{H}| = 2^{2^n}$. This is clear when we consider that, for fixed n, there are 2^n possible values of the input. A single hypothesis $h \in \mathcal{H}$ may classify each of these 2^n datapoints in one of two ways. Therefore, there are 2^{2^n} different ways to classify all 2^n points in the domain $\mathcal{H}$. Using Theorem 1, we can therefore bound the sample complexity of all four algorithms, though most interestingly the 1-nearest-neighbour algorithm in particular, by

$$m_{\mathcal{H}}(0.1, \delta) \leq 10 \log(|\mathcal{H}|/\delta) = 10(2^n - \log(\delta))$$

i.e. we have shown analytically one of the two bounds that we proposed for the 1-NN algorithm: $m(n) = O(2^n)$.

2.4

In order to derive an upper bound $\hat{p}_{m,n}$ on the probability that the perceptron will make a mistake on the sth example, after being trained on a dataset $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{s-1}, y_{s-1})$, we will use two theorems from the lecture notes.

Theorem 4: Suppose the data is drawn from a distribution ρ and a mistake bound for the algorithm $\mathcal{A}$ for any such set is B. Given $m \in \mathbb{Z}$, then let t be drawn uniformly at random from $\{1, \ldots, m\}$. Let $\mathcal{S}$ consist of t examples sampled i.i.d from ρ, and let $(\boldsymbol{x}', y')$ be an additional example sampled from ρ. Then,

$$\mathbb{P}(\mathcal{A}_{\mathcal{S}}(\boldsymbol{x}') \neq y') \leq \frac{B}{m}$$

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

This can be proven by simply considering that, by definition, there are no more than B trials for which the algorithm will make a mistake. Since t is drawn from $\{1, \ldots, m\}$, when the algorithm predicts on the $(t+1)$th, it has already seen up to m trials, up to B of which may have been mistakes. Therefore there is no more than a probability of B/m of making a mistake on the new datapoint.

It remains to find B for the perceptron in this setting. For this we use another theorem from the lecture notes:
Theorem 5: Consider the perceptron algorithm used in a learning setting such that the data is linearly separable by some margin γ, $\mathcal{Y} = \{-1, 1\}$, and there exists some constant $R > 0$ such that $\forall\, \boldsymbol{x}_t$: $\|\boldsymbol{x}_t\| \geq R$.
Then the number of mistakes that the perceptron algorithm can make is bounded by $\left(\frac{R}{\gamma}\right)^2$.

In the 'just a little bit' problem, we note that the classification of a point $\boldsymbol{x}$ depends wholly on whether $x_1 = -1$ or $x_1 = +1$. This means that the hyperplane separating the classes has margin $\gamma = 1$. Further, for the problem in n-dimensional space, we have that for all $\boldsymbol{x}_t \in \mathcal{X}$:

$$\|\boldsymbol{x}_t\| = \sqrt{\sum_{i=1}^{n} (\pm 1)^2} = \sqrt{n}$$

thus $R = \sqrt{n}$ satisfies the inequality $\|\boldsymbol{x}_t\| \geq R$ for all $\boldsymbol{x}_t$.

Therefore, for the 'just a little bit' problem, we obtain that $B = \left(\frac{R}{\gamma}\right)^2 = (\sqrt{n})^2 = n$. Therefore, our upper bound is

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

2.5

We now consider the **challenge** of finding a good lower bound of the sample complexity for the `1-nearest-neighbour` algorithm for the 'just a little bit' problem. we suggest this lower bound to be $f(n) = 2^n \log(n)$, so that $m = \Omega(f(n))$. Below we prove this bound. This proof is largely influenced by Section 19.2.1 of the book *Understanding Machine Learning*.[3]

We consider the problem of binary classification such that $\mathcal{Y} = \{-1, 1\}$, using 0-1 loss: $\ell(h(\mathbf{x}), y) = \mathbb{1}_{\{h(\mathbf{x}) \neq y\}}$.
Let $\mathcal{D}$ be the joint distribution on $\mathcal{X} \times \mathcal{Y}$, and let $\mathcal{D}_{\mathcal{X}}$ be the marginal distribution on $\mathcal{X}$.
For the 'just a little bit' problem, we define our input space to be $\mathcal{X} = \{-1, 1\}^n$, such that each point $\mathbf{x}^* \in \mathcal{X}$ is classified as $y^* := x_1^*$. Our training set, $S_x = (\mathbf{x}_1, \ldots, \mathbf{x}_m)$ is sampled uniformly on $\mathcal{X}$.
We define the function $\eta : \mathcal{X} \to [0, 1]$ to return the conditional probability of y given some specific $\mathbf{x}$:

$$\eta(\mathbf{x}) := \mathbb{P}(y = 1 | \mathbf{x})$$

We recognise that the value of y is deterministic when given some value of $\mathbf{x}$:

$$\eta(\mathbf{x}) = \begin{cases} 0; & x_1 = -1 \\ 1; & x_1 = 1 \end{cases}$$

We can also find the minimum value of $c > 0$ for which η is c-Lipschitz.

We wish to find $c > 0$ s.t. $\forall \mathbf{x}, \mathbf{x}' \in \mathcal{X}$,

$$|\eta(\mathbf{x}) - \eta(\mathbf{x}')| \leq c \|\mathbf{x} - \mathbf{x}'\|$$

As $\eta(\cdot) \in \{0, 1\} \Rightarrow |\eta(\mathbf{x}) - \eta(\mathbf{x}')| \leq 1$.
If $|\eta(\mathbf{x}) - \eta(\mathbf{x}')| = 1$, we require only that $x_1 \neq x_1'$, while the other coordinates of $\mathbf{x}, \mathbf{x}'$ could be the same. The minimum value of $\|\mathbf{x} - \mathbf{x}'\|$ for which $|\eta(\mathbf{x}) - \eta(\mathbf{x}')| = 1$ is therefore 2.

We therefore find that the minimal value of c must satisfy the inequality:

$$1 \leq 2c \Rightarrow c = \frac{1}{2}$$

This means that for all $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$,

$$|\eta(\mathbf{x}) - \eta(\mathbf{x}')| \leq \frac{1}{2} \|\mathbf{x} - \mathbf{x}'\|$$

[3] https://www.cs.huji.ac.il/ shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdfpage=61zoom=100,160,565

In this problem, we have defined the sample complexity to be

$$C(\mathcal{A}) := \min_{m \in \{1,2,\dots\}} \{\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq 0.1\}$$

We therefore look to investigate how we can bound the expected generalisation error, $\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})]$. We first note that $\mathcal{E}(\mathcal{A}_{\mathcal{S}_m}) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}_{\{h(\mathbf{x}\neq y\}}]$, therefore $\mathbb{E}_S[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})]$ is the probability that, having sampled a training set S and an additional sample $(\mathbf{x}, y)$, the label of the nearest neighbour to $\mathbf{x}$ in the training set is equal to y.
This means that we can find this probability in a few steps: first sample an input dataset of size m, $S_x \sim \mathcal{D}_{\mathcal{X}}^m$, and an additional unlabelled $\mathbf{x} \sim \mathcal{D}_{\mathcal{X}}$, then define $\pi_1(\mathbf{x})$ to be the nearest neighbour of $\mathbf{x}$ in S_x and sample $y \sim \eta(\mathbf{x}), y_{\pi_1(\mathbf{x})} \sim \eta(\pi_1(\mathbf{x}))$. This gives us:

$$\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] = \mathbb{E}_{S_x\sim\mathcal{D}_{\mathcal{X}}^m,\mathbf{x}\sim\mathcal{D}_{\mathcal{X}}}[\mathbb{P}_{y\sim\eta(\mathbf{x}),y'\sim\eta(\pi_1(\mathbf{x}))}[y \neq y']]$$

The probability inside the expectation can be written:

$$\begin{aligned}
\mathbb{P}_{y\sim\eta(\mathbf{x}),y'\sim\eta(\pi_1(\mathbf{x}))}[y \neq y'] &= \eta(\mathbf{x}')(1-\eta(\mathbf{x})) + (1-\eta(\mathbf{x}'))\eta(\mathbf{x}) \\
&= (\eta(\mathbf{x}) - \eta(\mathbf{x}) + \eta(\mathbf{x}'))(1-\eta(\mathbf{x})) \\
&\qquad + (1-\eta(\mathbf{x}) + \eta(\mathbf{x}) - \eta(\mathbf{x}'))\eta(\mathbf{x}) \\
&= 2\eta(\mathbf{x})(1-\eta(\mathbf{x})) + (\eta(\mathbf{x}) - \eta(\mathbf{x}'))(2\eta(\mathbf{x}) - 1)
\end{aligned}$$

As above, we have found that $\eta(\mathbf{x}) \in \{0, 1\}$, therefore $\eta(\mathbf{x})(1-\eta(\mathbf{x})) = 0$ and $|2\eta(\mathbf{x}) - 1| = 1$.
Thus, noting that $\mathbb{P}(\cdot) \geq 0$,

$$\begin{aligned}
\mathbb{P}_{y\sim\eta(\mathbf{x}),y'\sim\eta(\pi_1(\mathbf{x})}[y \neq y'] &= |(\eta(\mathbf{x}) - \eta(\mathbf{x}'))(2\eta(\mathbf{x}) - 1)| \\
&= |\eta(\mathbf{x}) - \eta(\mathbf{x}')| \\
&\leq \frac{1}{2}\|\mathbf{x} - \mathbf{x}'\|
\end{aligned}$$

Therefore, we obtain that

$$\begin{aligned}
\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] &= \mathbb{E}_{S\sim\mathcal{D}^m,\mathbf{x}\sim\mathcal{D}}[\mathbb{P}_{y\sim\eta(\mathbf{x}),y'\sim\eta(\pi_1(\mathbf{x})}(y \neq y')] \\
&\leq \frac{1}{2}\mathbb{E}_{S\sim\mathcal{D}^m,\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x} - \mathbf{x}_{\pi_1(\mathbf{x})}\|]
\end{aligned}$$

The next step is to bound the expected distance between a random $\mathbf{x}$ and its nearest neighbour $\mathbf{x}_{\pi_1(\mathbf{x})}$:
Let $C_1, C_2, \ldots, C_r$, $r = 2^n$, be a collection of singleton subsets of $\mathcal{X}$ such that each C_i contains exactly one unique point in $\mathcal{X}$. It is clear that $\mathcal{X} = \bigcup_{i=1}^{r} C_i$. We look to bound the probability that a randomly drawn $\mathbf{x}$ does not appear in our training set. This is the expectation given by:

$$\mathbb{E}_S\Big[\sum_{i:C_i\cap S=\emptyset} \mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i]\Big] = \sum_{i=1}^{r} \mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i]\cdot \mathbb{E}_S[\mathbb{1}_{[C_i\cap S=\emptyset]}]$$

For each i, we have that

$$\mathbb{E}_S[\mathbb{1}_{[C_i\cap S=\emptyset]}] = \mathbb{P}_S[C_i\cap S=\emptyset] = (1-\mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i])^m \leq e^{-\mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i]m}$$

Therefore:

$$\mathbb{E}_S\Big[\sum_{i:C_i\cap S=\emptyset} \mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i]\Big] \leq \sum_{i=1}^{r} \mathbb{P}_{\mathbf{x}\sim\mathcal{D}_x}[\mathbf{x}\in C_i]\cdot e^{-\mathbb{P}_{\mathbf{x}\sim\mathcal{D}_x}[\mathbf{x}\in C_i]m}$$

We now recognise that, as mentioned at the start of the question, the values $\mathbf{x}$ are drawn uniformly from the points in $\mathcal{X}$. As C_i is a collection of unique singleton sets that cover $\mathcal{X}$, we have that, for all i, $\mathbb{P}_{\mathbf{x}\sim\mathcal{D}_x}[\mathbf{x}\in C_i] = \frac{1}{2^n}$. Thus,

$$\begin{aligned}\mathbb{E}_S\Big[\sum_{i:C_i\cap S=\emptyset} \mathbb{P}_{\mathbf{x}}[\mathbf{x}\in C_i]\Big] &\leq \sum_{i=1}^{2^n} \mathbb{P}_{\mathbf{x}\sim\mathcal{D}_x}[\mathbf{x}\in C_i]\cdot e^{-\mathbb{P}_{\mathbf{x}\sim\mathcal{D}_x}[\mathbf{x}\in C_i]m}\\ &= 2^n\frac{1}{2^n}\cdot e^{-\frac{m}{2^n}} = e^{-\frac{m}{2^n}}\end{aligned}$$

This inequality will allow us to bound the expected distance between $\mathbf{x}$ and its nearest neighbour in S, $\mathbf{x}_{\pi_1(\mathbf{x})}$,

$$\mathbb{E}_{S\sim\mathcal{D}^m,\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\|] = \mathbb{E}_{S\sim\mathcal{D}^m}[\mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\|]]$$

Let $\mathbf{x}_{\pi_1(\mathbf{x})} \in C_{i^*}$. If $\mathbf{x}\in C_{i^*} \Rightarrow \|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\| = 0$.

If $\mathbf{x}\notin C_{i^*}$, then we can only bound by the maximum possible distance between any two points in $\mathcal{X}$. This occurs if $\mathbf{x}' = -\mathbf{x}$, in which case

$$\|\mathbf{x}-\mathbf{x}'\| = \sqrt{\sum_{i=1}^{n}(x_i-x_i')^2} = \sqrt{\sum_{i=1}^{n}2^2} = \sqrt{4n} = 2\sqrt{n}$$

Thus, if $\mathbf{x}\notin C_{i^*} \Rightarrow \|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\| \leq 2\sqrt{n}$.
This means that we can split the expectation $\mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\|]$ into two possible cases:
Case 1: $\mathbf{x} = \mathbf{x}_{\pi_1(\mathbf{x})}$. This is equivalent to the case that $\mathbf{x}\in\bigcup_{i:C_i\cap S\neq\emptyset} C_i$ (in which case, $\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\| = 0$).
Case 2: $\mathbf{x}\neq\mathbf{x}_{\pi_1(\mathbf{x})}$. This is equivalent to the case that $\mathbf{x}\in\bigcup_{i:C_i\cap S=\emptyset} C_i$ (in which case, $\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\| \leq 2\sqrt{n}$).
i.e.

$$\begin{aligned}\mathbb{E}_{S\sim\mathcal{D}^m}[\mathbb{E}_{\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\|]] &\leq \mathbb{E}_{S\sim\mathcal{D}^m}\Big[\mathbb{P}\Big[\mathbf{x}\in\bigcup_{i:C_i\cap S\neq\emptyset} C_i\Big]\cdot 0 + \mathbb{P}\Big[\mathbf{x}\in\bigcup_{i:C_i\cap S=\emptyset} C_i\Big]\cdot 2\sqrt{n}\Big]\\ &= \mathbb{E}_{S\sim\mathcal{D}^m}\Big[\mathbb{P}\Big[\mathbf{x}\in\bigcup_{i:C_i\cap S=\emptyset} C_i\Big]\cdot 2\sqrt{n}\Big]\\ &= 2\sqrt{n}\cdot\mathbb{E}_{S\sim\mathcal{D}^m}\Big[\sum_{i:C_i\cap S=\emptyset}\mathbb{P}\Big[\mathbf{x}\in C_i\Big]\Big]\\ &\leq 2\sqrt{n}e^{-\frac{m}{2^n}}\end{aligned}$$

where we have used the bound that we derived earlier.

We can then put all of these results together to obtain the following:

$$\begin{aligned}
\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] &\leq \frac{1}{2}\mathbb{E}_{S\sim\mathcal{D}^m,\mathbf{x}\sim\mathcal{D}}[\|\mathbf{x}-\mathbf{x}_{\pi_1(\mathbf{x})}\|] \\
&\leq \frac{1}{2}2\sqrt{n}e^{-\frac{m}{2^n}} \\
&= \sqrt{n}e^{-\frac{m}{2^n}} \\
\Rightarrow \mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] &\leq \sqrt{n}e^{-\frac{m}{2^n}}
\end{aligned}$$

We have defined the sample complexity to be the smallest value of m for which $\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq 0.1$. We can find this m as a function of n by setting:

$$\begin{aligned}
\mathbb{E}_{S\sim\mathcal{D}^m}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq \sqrt{n}e^{-\frac{m}{2^n}} &\leq 0.1 \\
\Rightarrow e^{\frac{m}{2^n}} &\geq 10\sqrt{n} \\
\Rightarrow \frac{m}{2^n} &\geq \log(10\sqrt{n}) \\
\Rightarrow m &\geq 2^n\log(10\sqrt{n}) \\
&= 2^n\log(10) + 2^n\log(\sqrt{n})
\end{aligned}$$

Therefore $2^n\log(10) + 2^n\log(\sqrt{n})$ is a lower bound on the value of m required to achieve an expected generalisation error ≤ 0.1:

$$\begin{aligned}
m &= \Omega(2^n\log(10) + 2^n\log(\sqrt{n})) \\
&= \Omega(2^n\log(\sqrt{n})) \\
&= \Omega(\frac{1}{2}2^n\log(n)) \\
&= \Omega(2^n\log(n))
\end{aligned}$$

Thus, we suggest a 'good' lower bound for the sample complexity of the `1-nearest-neighbour` algorithm on the 'just a little bit' problem to be $f(n) = 2^n\log(n)$, so that $m = \Omega(f(n))$.