Supervised Learning (COMP0078) Coursework 2

Charita Dellaporta (19025680) & Callum Kai Lau (19102521)

January 2020

1 PART I

1.1 Introduction

The introductory part of this report will discuss the methods used to implement the Multi-class Kernel Perceptron.

1.1.1 The Two Class Kernel Perceptron

A description of the two class kernel perceptron algorithm described in the assignment is given here as a reference point. This will help us to build upon and compare the more advanced implementations of the kernel perceptron that we develop later.

Two Class Kernel Perceptron	
Input	$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\} \in (\mathcal{R}^n, \{-1, +1\})^m$
Initialisation	$\mathbf{w}_0 = \mathbf{0} \quad (\alpha_0 = 0)$
Prediction	Upon receiving the tth instance $\mathbf{x}_t$ predict $\hat{y} = sign(\mathbf{w}_t(\mathbf{x}_t)) = sign(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$
Update	if $\hat{y}_t = y_t$ then $\alpha_t = 0$ else $\alpha_t = y_t$ $\mathbf{w}_{t+1}(\cdot) = \mathbf{w}_t(\cdot) + \alpha_t K(\mathbf{x}_t, \cdot)$

1.1.2 Generalising to K Classes

The preliminary method used to generalise the kernel perceptron so that it may classify multiple classes was the one-vs-all strategy. Instead of a single classifier $\mathbf{w}$, the learner L now uses a classifier per class $\mathbf{w}^{(k)}$, $k \in (1, \ldots, K)$. Rather than receiving a single class label $\hat{y} \in (-1, +1)$ for a decision, a set of 'confidence scores' $s^{(k)}$ per classifier is used instead. The classification of a test example is through the label k of the classifier $\mathbf{w}^{(k)}$ with the highest confidence score $\text{argmax}_{k \in \{1,\ldots,K\}} s^{(k)}$. A description of the algorithm given below. Firstly the necessary data structures are described:

1. Input data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\} \in (\mathcal{R}^n, \{1, \ldots K\})^m$.

2. The gram matrix $G \in \mathcal{R}^{(m \times m)}$ consisting of the kernel $K(\cdot, \cdot)$ applied to all the training points $\mathbf{x}_i$ - i.e. $(G)_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$.[1]

3. The classifier matrix $W \in \mathcal{R}^{(K \times m)}$ consisting of the per-class classifiers $\mathbf{w}^{(k)}$, $k \in \{1, \dots K\}$ - i.e. we have $(W)^{\top}_{k\cdot} = \mathbf{w}^{(k)}$. Note that a classfier $\mathbf{w}^{(k)}$ is parametrised entirely by the weights $\alpha_t^{(k)}$ it places on each training example $\mathbf{x}_t$ for $t \in \{1, \dots, m\}$, and therefore W is simply a matrix consisting of the weights each classifier assigns to each training example.

Multi-class Kernel Perceptron (training)	
Input	Gram matrix G, classifier matrix W, sequence of training examples $(\mathbf{x}_t, y_t) \in (\mathcal{R}^n, \{1, \dots K\})$, mistake count M.
Initialisation	Set $W_{kt} = 0$ for all k, t, M $= 0$
Prediction	Compute the scores $s^{(k)}$ for each of the K classifiers: $s^{(k)} = \sum_{i=0}^{t-1} \alpha_i^{(k)} K(\mathbf{x}_i, \mathbf{x}_t) = \sum_{i=0}^{t-1} W_{ki} G_{it}$. Note that we have $W_{ki} = 0$ for $i \geqslant t$ and therefore we can compute the scores $\mathbf{s}$ for all K classifiers in one quick matrix multiplication: $\mathbf{s} = W[G]_{\cdot t}$
Update	for each classifier $\mathbf{w}^{(k)}$ for $k \in \{1, \dots, K\}$: 1. If the training label y_t is the same as the classifier label k: $y_t = k$, then set a new label $\hat{y} = +1$, else set $\hat{y} = -1$. 2. If $\hat{y} * s^{(k)} \leqslant 0$ then we have misclassified for this classifier. Therefore set $W_{kt} = W_{kt} - sign(s^{(k)})$ The final classification for the example $\mathbf{x}_t$ is the label $\hat{k}$ of the classifier $\mathbf{w}^{(k)}$ with the highest confidence score $\hat{k} = \text{argmax}_{k \in \{1, \dots, K\}} s^{(k)}$. If $\hat{k} \neq y_t$, then we have a mistake and we update the mistake counter $M = M + 1$

Note that the one-vs-all multi-class perceptron is very similar to the two-class perceptron. In a sense, we use K binary perceptrons and apply the binary perceptron algorithm to each perceptron separately. The only differences which arise are :

· For any example $\mathbf{x}_t$ we now have K different possible labellings y_t can take, and so during classification we have $K - 1$ negative (wrong) labellings and 1 positive (correct) labelling. Therefore for each classifier $\mathbf{w}^{(k)}$ we can use the binary perceptron algorithm by simply converting the label y_t of the example to $+1$ if the labelling is the same as the classifier label (a positive example) or to -1 otherwise (a negative example). This is the purpose of the temporary label $\hat{y}$ in the algorithm described above.

· Classification of a test point $\mathbf{x}^*$ no longer depends simply upon the sign of the single classifier applied to the test point but instead upon the label $\hat{k}$ of the classifier $\mathbf{w}^{(\hat{k})}(\mathbf{x}^*)$ which produces the highest 'confidence score' s^k.

[1] Reasons for forming the Gram matrix are given in Section 1.1.3

1.1.3 Evaluating the sum $\mathbf{w}(\cdot)$

In this section, some discussion of how the sum $\mathbf{w}(\cdot)$ was represented and evaluated is given. Some details were mentioned in the preceding section, however we repeat them here.

Representation

Since $\mathbf{w}^{(k)}(\cdot) = \sum_{i=1}^{m} \alpha_i^{(k)} K(\mathbf{x}_i, \cdot)$ we can represent the sum using two matrices: W consisting of the classifiers $\mathbf{w}^{(k)}(\cdot)$ and G which is the gram matrix formed by the kernel (polynomial or Gaussian) applied to the training points $\mathbf{x}_t$, i.e. $[G]_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$. As noted before, each classifier can be entirely parametrised by the 'weights' $\alpha_t^{(k)}$ placed on each training example, and therefore the matrix W is specified by $[W]_{kt} = \alpha_t^{(k)}$. Additionally, for the training samples we chose to pre-compute the gram matrix G, as opposed to on-the-fly when a new term was added to the sum during training. The primary reason for this was implementation speed.

Evaluation

Since we have preformed the gram matrix G, during training we can evaluate the sum $\mathbf{w}_t^{(k)}(\mathbf{x}_t) = \sum_{i=1}^{t} \alpha_i^{(k)} K(\mathbf{x}_i, \mathbf{x}_t)$ using a single matrix multiplication. In practice, we can multiply the k'th row of the classifier matrix $[W]_{k\cdot}$ by the t'th column of the gram matrix $[G]_{\cdot t}$ to find the sum $\mathbf{w}_t^{(k)}(\mathbf{x}_t) = [W]_{k\cdot} \cdot [G]_{\cdot t}$. Therefore to evaluate the sum for all classifiers (denoted $\mathbf{s}$ as introduced in the previous section), we can use instead the entire classifier matrix: $\mathbf{s} = W \cdot [G]_{\cdot t}$. In this sense, no terms $\mathbf{x}_t$ are 'added' to the sum during training, and instead we update the classifiers by selecting the t'th column of the gram matrix instead. This greatly improved implementation speed because the code then becomes vectorised. For the polynomial kernel and input data $\mathbf{x}$, the gram matrix could be formed efficiently by computing $\mathbf{x}\mathbf{x}^\top$ and raising this to the polynomial degree d. The sums $\mathbf{s}$ could then be calculated via a matrix product. This proved to be much quicker than representing $\mathbf{w}(\cdot)$ as an expanding list and computing each $K(\mathbf{x}_i, \mathbf{x}_t)$ on-the-fly due to the benefits of vectorisation and the inefficiency of indexing.

1.1.4 Number of Epochs

The number of epochs required to train the polynomial Kernel perceptron with $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q})^d$ is an important choice in order to find the rate of convergence of the classifiers. We also wish to find a suitable number of epochs so that the classifiers do not *overfit* to the data. Therefore we performed some preliminary analysis of the data in order to find the minimum number of epochs so that the classifiers have trained sufficiently on the data but have not overfitted. The method used was - for each d train on 80% of the data for a large number of epochs (30) and test on the remaining 20% to find the test error. Then average over multiple runs (5) to find the minimum number of epochs at which the test error for all d no longer decreases consistently. We then use this minimum as the number of epochs for subsequent questions.

A graph of the test error against epoch for each d is given below:

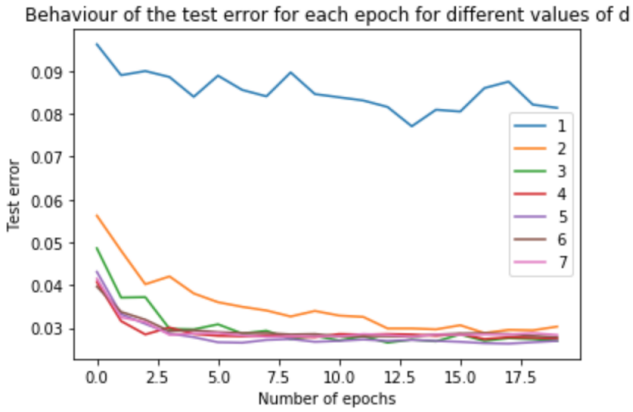


Figure 1: Test error against epochs, for $d = 1, \ldots, 7$

This graph provides strong evidence that for $d \neq 1, 2$, the test error seems to converge within 5 epochs. In the interest of simplicity and computational speed, therefore, we decide to use 5 epochs for training for all values of d in the kernel perceptron. Although we may sacrifice some test error accuracy for $d = 1, 2$, it seems that limiting the number of epochs to 5 for the other values of d will provide enough time for convergence and limit the potential for overfitting.

1.2 Part a - Basic Results

For the polynomial Kernel $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q})^d$, we perform 20 runs for d =1, ..., 7 splitting the data into random 80% training and 20% test sets. The reported mean test and training errors as well as their standard deviations are given in the table below:

	Training error	Test error
1	7.96% ± 0.27%	9.37% ± 1.54%
2	0.90% ± 0.08%	3.68% ± 0.45%
3	0.27% ± 0.06%	3.13% ± 0.36%
4	0.13% ± 0.04%	2.92% ± 0.83%
5	0.08% ± 0.03%	2.77% ± 0.35%
6	0.07% ± 0.02%	2.92% ± 0.37%
7	0.06% ± 0.02%	2.89% ± 0.32%

Figure 2: Mean test and training error

1.3 Part b - Cross Validation

We performed 5-fold cross validation using the scikit-learn function `sklearn.model_selection.KFold` to randomly split the 80% training data into 5 random train/test folds to find the best d^* over 20

runs. The best d^* for each run along with its test error after retraining on the full 80% training set is given below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
d*	6	6	7	4	5	5	7	7	7	5	3	5	3	6	6	5	6	6	4	5
test error	2.42%	2.85%	2.90%	2.80%	3.12%	3.06%	2.42%	2.74%	2.47%	3.12%	3.55%	2.42%	2.42%	4.09%	2.37%	2.53%	2.69%	2.37%	2.58%	2.53%

Figure 3: Best d^* for each run

Furthermore we provide the mean d^* and test error over the 20 runs:

	Mean d* and std	Mean test error and std
1	5.40 ± 1.20	2.77% ± 0.43%

Figure 4: Mean d^* and test error

1.4 Part c - Confusion Matrix

In producing the confusion matrix we used the exact same method as for cross-validation except a simple tally was kept for every misclassification during the retraining and testing phase for the best d^*. If a misclassification was made for a test point with true label i and the classifier predicted j we added one to the count in the confusion matrix C at C_{ij}.

	0	1	2	3	4	5	6	7	8	9
0	0.00 ± 0.00	0.20 ± 0.40	0.55 ± 0.74	0.40 ± 0.66	0.45 ± 0.59	0.70 ± 0.84	1.05 ± 0.97	0.20 ± 0.51	0.15 ± 0.48	0.05 ± 0.22
1	0.15 ± 0.48	0.00 ± 0.00	0.20 ± 0.40	0.00 ± 0.00	1.30 ± 4.14	0.00 ± 0.00	0.40 ± 0.58	0.10 ± 0.30	0.15 ± 0.36	0.45 ± 1.32
2	0.85 ± 0.65	0.40 ± 0.66	0.00 ± 0.00	1.25 ± 1.04	1.20 ± 1.03	0.20 ± 0.40	0.25 ± 0.43	1.10 ± 1.22	0.75 ± 0.99	0.05 ± 0.22
3	0.25 ± 0.43	0.20 ± 0.51	1.10 ± 0.99	0.00 ± 0.00	0.10 ± 0.30	3.05 ± 1.63	0.05 ± 0.22	0.80 ± 1.03	1.35 ± 1.49	0.40 ± 0.66
4	0.20 ± 0.40	0.75 ± 0.89	0.70 ± 0.71	0.05 ± 0.22	0.00 ± 0.00	0.20 ± 0.40	0.85 ± 0.73	0.40 ± 0.49	0.10 ± 0.30	1.45 ± 1.16
5	1.15 ± 0.91	0.10 ± 0.30	0.70 ± 0.71	1.50 ± 1.69	0.75 ± 0.94	0.00 ± 0.00	1.25 ± 1.09	0.10 ± 0.30	0.55 ± 0.97	0.65 ± 0.85
6	0.80 ± 0.81	0.35 ± 0.65	0.55 ± 0.74	0.00 ± 0.00	1.00 ± 1.10	0.45 ± 0.59	0.00 ± 0.00	0.05 ± 0.22	0.20 ± 0.51	0.00 ± 0.00
7	0.05 ± 0.22	0.25 ± 0.70	0.65 ± 0.79	0.05 ± 0.22	1.35 ± 1.19	0.25 ± 0.43	0.00 ± 0.00	0.00 ± 0.00	0.35 ± 0.57	1.10 ± 0.99
8	1.00 ± 0.89	0.60 ± 0.80	0.80 ± 0.81	1.80 ± 1.25	0.85 ± 0.73	1.60 ± 1.36	0.40 ± 0.97	0.75 ± 0.70	0.00 ± 0.00	0.45 ± 0.80
9	0.35 ± 0.48	0.15 ± 0.36	0.35 ± 0.48	0.25 ± 0.54	1.90 ± 1.81	0.25 ± 0.43	0.10 ± 0.30	1.00 ± 0.95	0.20 ± 0.40	0.00 ± 0.00

Figure 5: Confusion Matrix

1.5 Part d - Hardest to Predict

The hardest to predict digits are the ones that are most frequently misclassified. To find the hardest to predict we split the entire dataset into 10-folds using `sklearn.model_selection.KFold`. For every d values $d = 5, 6, 7$, we trained the classifier for 5 epochs on the 9-fold training set and tested on the remaining test fold. For all 10 test folds we counted the number of mistakes made on each

test example, and added this to a total tally of mistakes for every data point. This procedure was repeated over 40 runs and the digits with the 5 highest mistake counts are given as the 5 hardest to predict. The motivation for performing K-fold splits instead of simply creating a random 80/20 training/test split and performing multiple runs was to ensure that each data point was tested on the exact same number of times - this ensures that the digits misclassified most frequently are less subject to the random process of placing them into the test sets with varying frequencies. Furthermore a 10-fold split was used with the reasoning being that the harder to predict digits would be misclassified more often than the 'not-so-hard' digits if a larger training set was used. Finally the intuition behind using the better performing d values was similar in that a poor value of d (such as $d = 1$) would more frequently misclassify the not-so-hard digits, making it harder to separate out the harder digits.

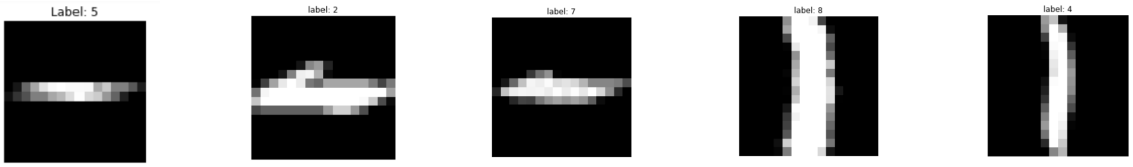


Figure 6: Hardest to predict digits

1.6 Part e - Gaussian Kernel

Following the experimentation with the polynomial kernel we proceeded by repeating the experiment using the Gaussian kernel

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

In order to compute the gram matrix in this case we used the 'euclidean distances' function given by the 'sklearn.metrics.pairwise' library to first compute the pairwise squared distances between all points and store them in a matrix. In this library the *squared* distance between two vectors x,y is computed as

$$dist(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{x}) - 2 * (\mathbf{x} \cdot \mathbf{y}) + (\mathbf{y} \cdot \mathbf{y})$$

This method is very computationally efficient, however is not as precise as directly calculating the euclidean distance, which may lead to small computational errors. For example, it might produce a gram matrix which is not completely symmetric as we would expect. However, this method significantly speeds up our experiments with small computational cost and helps the better analysis of the algorithm with the Gaussian kernel. We then proceeded by applying the multiplication by -c and then raising it to the exponential function pointwise in the matrix, using the scipy library. This method ended up being significantly more efficient than computing each term of the matrix separately.

Following this, we performed some initial experiments in order to decide on a suitable set S of values of c to cross validate over, where c is essentially the band width of the kernel. We begin by trying multiple values of c ranging from 0.0001 to 100 and observed that the model performs

the best when c small and in particular between 0.001 and 0.08 so decided to cross validate over 7 equidistant values in this range.

Another important choice was the number of epochs used during training. We used the same method as in section 1.1.4 in order to come up with a suitable number of epochs for which we observe convergence of the training error but avoid overfitting on the training data. Plotting the test error against epochs for $c = 0.001, \ldots, 0.08$ we obtain the following graph:

Figure 7: Test error against epochs, for $c = 0.001, \ldots, 0.08$

We can observe that ideally we would have to use around 15 epochs in order to ensure that the test error converges for all values of c. However, this is extremely computationally expensive and it would require a very large amount of time. Thus, noting that for most values of c the test error converges after around 5 epochs we thought it is reasonable to trade off accuracy with computational efficiency and continue our analysis using 5 epochs for each run, which can be achieved in a reasonable amount of time. We now proceed to repeat parts a and b as above.

1.6.1 Repeat Part a

For the Gaussian Kernel $K(\mathbf{p}, \mathbf{q}) = e^{-c||\mathbf{p}-\mathbf{q}||^2}$ we perform 20 runs for $c = 0.001, \ldots, 0.08$ splitting the data into random 80% training and 20% test sets. The reported mean test and training errors as well as their standard deviations are given in the table below:

	Kernel width	Training error	Test error
1	0.001000	6.22% ± 0.24%	7.55% ± 1.96%
2	0.014167	0.05% ± 0.02%	2.65% ± 0.37%
3	0.027333	0.03% ± 0.02%	2.71% ± 0.40%
4	0.040500	0.03% ± 0.02%	3.44% ± 0.38%
5	0.053667	0.02% ± 0.02%	4.03% ± 0.52%
6	0.066833	0.02% ± 0.02%	4.42% ± 0.40%
7	0.080000	0.02% ± 0.02%	4.59% ± 0.60%

Figure 8: Mean test and training error using the Gaussian kernel

1.6.2 Repeat Part b

We perform cross validation over the values of c in order to obtain the value for which we achieve the best test error for each run. We then retrain on the full 80% training set and report the test errors on the rest 20% of the data. The best value of c^* together with the mean test error and its standard deviation for each such c^* are displayed in the tables below:

	c*	test error
1	0.027333	3.06%
2	0.014167	2.37%
3	0.014167	2.15%
4	0.014167	2.15%
5	0.014167	2.69%
6	0.014167	2.85%
7	0.014167	2.63%
8	0.014167	3.23%
9	0.014167	3.28%
10	0.014167	2.42%
11	0.014167	3.12%
12	0.014167	2.96%
13	0.014167	2.53%
14	0.014167	2.63%
15	0.014167	3.60%
16	0.014167	2.96%
17	0.014167	2.53%
18	0.014167	2.69%
19	0.027333	3.06%
20	0.027333	2.53%

(a) Gaussian Kernel best c and test error

	Mean c* and std	Mean test error and std
1	0.0161 ± 0.0047	2.77% ± 0.37%

(b) Gaussian Kernel mean c and test error

Figure 9: Cross Validation results for the Gaussian Kernel

We observe that the Gaussian Kernel achieves a mean test error equal to 2.77% which is the same value to the mean test error obtained from the cross validation using the polynomial kernel. The standard deviation at 0.37% was slightly lower than for the polynomial kernel, however no significant conclusions can be made as to whether the Gaussian kernel outperforms the polynomial kernel based on the data we found. Therefore it seems that there is no significant advantage to representing the digits in Radial Basis Function space for the Perceptron algorithm.

1.7 Part f - One vs One

The one vs all strategy above performed well on our data set however it may suffer if the data set is unbalanced, that is, contains a large number of data points for one class and a very small amount of data for another class thus having a much lower confidence score for some binary classifiers than others. A popular alternative to this method is the one vs one strategy which instead of learning K binary classifiers (one for each class) it learns $\binom{K}{2} = \frac{K(K-1)}{2}$ binary classifiers, one for each pair of classes. This means, that for any two classes $i, j \in \{1, \ldots, k\}, i \neq j$ the corresponding classifier is being trained only on those data points which have labels i or j. Hence, each classifier is individually very strong on classifying between two particular classes. Once all $\frac{K(K-1)}{2}$ classifiers have been trained, we make predictions by taking the majority vote of all the binary classifiers. This means, that for a test point $(\mathbf{x}^*, \mathbf{y}^*)$, each classifier (i,j) predicts between classes i and j and we hence have $\frac{K(K-1)}{2}$ different predictions for $\mathbf{y}^*$. We take as our final prediction the class which has the most votes. As in the one-vs-all case we denote again:

1. Input data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\} \in (\mathcal{R}^n, \{1, \ldots K\})^m$.
2. The gram matrix $G_{k,l} \in \mathcal{R}^{(m_{k,l} \times m_{k,l})}$, for each pair of classes $k, l \in \{1, \ldots, K\}$. That is, we have $\frac{K(K-1)/K}{2}$ such matrices G, one for each pair of classes k,l and $m_{k,l}$ is equal to the number of training points in our training set which have either label k or l. Each entry of such matrix $G_{k,l}$ consists of the kernel $K(\cdot, \cdot)$ applied to all $m_{k,l}$ training points $\mathbf{x}_i$ - i.e. $(G_{k,l})_{ij} = K(\mathbf{x}_i, \mathbf{x}_j), i \in \{1, \ldots, m_{k,l}\}$
3. The classifier matrix $W \in \mathcal{R}^{(\frac{K(K-1)}{2} \times m)}$ consisting of the binary classifiers $\mathbf{w}^{(k)}$, $k \in \{1, \ldots \frac{K(K-1)}{2}\}$ - i.e. we have $(W)^\top_{k\cdot} = \mathbf{w}^{(k)}$. As before W is simply a matrix consisting of the weights each classifier assigns to each training example.
4. The votes matrix $V \in \mathcal{R}^{(\frac{K(K-1)}{2} \times m)}$, consisting of the prediction $\hat{y}_{k,m}$ of each classifier k for the mth example.

The pseudo-code for this method is described below:

One-vs-One Multi-class Kernel Perceptron (training)	
Input	Gram matrices $G_{k,l}$, classifier matrix W, sequence of training examples $(\mathbf{x}_t, y_t) \in (\mathcal{R}^n, \{1, \dots K\})$, votes matrix V, mistake count M.
Initialisation	Set $W_{kt} = 0$ for all k, t, $v_i = 0$ for all i, M = 0
Prediction	For each of the $\frac{K(K-1)}{2}$ binary classifiers: for $k \in \{1, \dots, \frac{K(K-1)}{2}\}$, • Get subset of the training set of size $m_{k,l}$ consisting of these training points with labels k and l. • Upon receiving the tth instance $\mathbf{x}_t$ predict $\hat{y} = sign(\mathbf{w}_k t(\mathbf{x}_t)) = sign(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t))$
Update	For each classifier $\mathbf{w}^{k,l}$ for $k, l \in \{1, \dots, \frac{K(K-1)}{2}\}$: For a true label y_t set $y_{real} = 1$ if $y_t = k$ and $y_{real} = -1$ if $y_t = l$ If $\hat{y}_t \neq y_{real}$ then $\alpha_t = y_{real}$ For the final prediction compute $V_{k,t} = \hat{y}_k$, where $\hat{y}_k$ is the prediction of the kth classifier. The final classification for the example $\mathbf{x}_t$ is $\hat{k} = \text{argmax}_{k \in \{1,\dots,K\}} V_{kt}$, i.e. the most voted class among all classifiers for example t. If $\hat{k} \neq y_t$, then we have a mistake and we update the mistake counter $M = M + 1$

1.7.1 Repeat part a

We move on into implementing the one vs one generalisation to K classes as discussed above. Firstly, we explore the number of epochs needed to achieve convergence of the test error and avoid overfitting on the training data. This was done based on the method discussed in section 1.1.4 and the results obtained are displayed in the graph below:

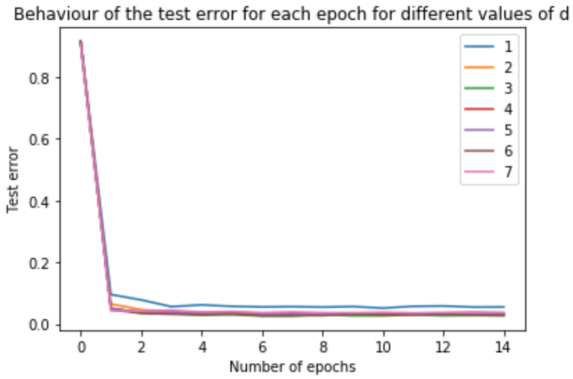


Figure 10: Test error against epochs, for $d = 1, \dots, 7$

In the graph of epochs against test error above we observe that the test error converges very quickly, after 3 epochs for all values of d.

Hence we proceed by performing 20 runs using the one vs one method with the Polynomial Kernel for $d = 1, \ldots, 7$ splitting the data into random 80% training and 20% test sets. The reported mean test errors as well as their standard deviations are given in the table below:

	Test error
1	6.31% ± 0.65%
2	4.29% ± 0.51%
3	3.71% ± 0.43%
4	3.45% ± 0.46%
5	3.36% ± 0.41%
6	3.35% ± 0.43%
7	3.64% ± 0.44%

Figure 11: Test error using the one-vs-one method

1.7.2 Repeat part b

As previously, we perform cross validation over the values of d and report the best value of d obtained for each d as well as the mean d^* and mean test error obtained from re-training on the full 80% of the data and testing on the rest 20%, using the best value of d obtained in each run.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
d*	6	5	4	7	5	6	5	4	4	5	5	5	6	5	4	6	5	5	5	6
test error	3.87%	3.17%	3.06%	3.23%	3.12%	3.28%	3.76%	3.23%	3.87%	4.03%	3.49%	3.49%	2.85%	3.60%	3.06%	2.85%	2.90%	3.39%	3.60%	3.12%

Figure 12: Best d^* for each run

	Mean d* and std	Mean test error and std
1	5.15 ± 0.79	3.35% ± 0.35%

Figure 13: Mean d^* and test error

1.8 Part g - Alternate Algorithms

1.8.1 Forgetron

Despite performing well on online classification tasks, kernelised perceptrons require unbounded memory used to store the online hypothesis. Additionally, since all previous training examples are stored in memory the running time of each online round scales linearly $O(m)$ with each additional training example. One possible solution that presents its self is a kernel-based perceptron on a *budget* - the forgetron. The idea is to keep a budget $\mathcal{B}$ which is a fixed limit on the number of training examples stored by removing support vectors from the kernel expansion. Let the training set or *Cache* held in memory be denoted $\mathcal{S}$. Therefore upon receiving example $\mathbf{x}_t$, if $|\mathcal{S}| > \mathcal{B}$ we discard a sample $\mathbf{x}^* \in \mathcal{S}$ according to some criteria.

Example criteria include, for instance, simply the oldest example or a random example. Mistake bounds for binary kernel perceptrons using similar criteria during an additional removal step have been proved in 'The Forgetron: A Kernel-Based Perceptron on a Budget'[2]. However we will use the removal criteria suggested in 'Online Classification on a Budget'[3] and specifically page 5 of 'Fast Kernel Classifiers with Online and Active Learning'[4]. We scan the cache $\mathcal{S}$ for seemingly redundant examples by examining their margins. Essentialy, examples with a large margin contribute little to the hypothesis and so are most suitable for removal. For the binary perceptron this is given by:

$$\underset{i \in \mathcal{S}}{\operatorname{argmax}}\, y_i(\mathbf{w}(\mathbf{x}_i) - \alpha_i K(\mathbf{x}_i, \mathbf{x}_i)) \tag{1}$$

These values can be seen as distances and so we aim to discard examples with the maximum distance. However, we must generalise this to the multi-class case where the labels $y_i \in \{1, \ldots, K\}$. The method is analogous to extending the classifier to hold K binary perceptrons capable of distinguishing only their respective class k, i.e. whether $\mathbf{x}_t \in k$ or $\mathbf{x}_t \notin k$. We give each perceptron a total budget $\mathcal{B}$ and hold a set of K caches $S^{(k)}$ $k \in \{1, \ldots, K\}$ for each perceptron. Whenever a single perceptron $\mathbf{w}^{(k)}$ misclassifies an example we add this example to its cache: $\mathcal{S}^{(k)} \leftarrow \mathcal{S}^{(k)} \cup \{\mathbf{x}_t\}$. If the total size of the Caches (i.e. number of training examples held in memory) exceeds the budget limit $\bigcup_{k=1}^{K} |\mathcal{S}^k| > \mathcal{B}$, we remove an example via the rule:

$$\underset{i,k \in \bigcup_{k=1}^{K} \mathcal{S}_i^{(k)}}{\operatorname{argmax}} \hat{y}_i(\mathbf{w}^{(k)}(\mathbf{x}_i) - \alpha_i^{(k)} K(\mathbf{x}_i, \mathbf{x}_i)) \tag{2}$$

Where $\mathcal{S}_i^{(k)}$ indexes the *i'th* example in the *k'th* Cache. Two modifications to the original Multi Class Perceptron algorithm are required. In the **Prediction** step only the training examples in the cache $\mathbf{x}_t \in \mathcal{S}^{(k)}$ are used in the kernel expansion. In the **Update** step we must now add and remove terms to the Caches $\mathcal{S}^{(k)}$ according to the criterion (2) *on-the-fly*. However, as a result we are now unable to vectorise the code, and so update the notation accordingly. The updated algorithm is given below:

[2] Shai Shalev-Shwartz et al. `https://ttic.uchicago.edu/~shai/papers/DekelShSi07.pdf`

[3] Yoram Singer et al. `https://papers.nips.cc/paper/2385-online-classification-on-a-budget.pdf`

[4] Antoine Bordes et al. `http://www.robots.ox.ac.uk/~vgg/sorg/huller3.pdf`

Multi-class Forgetron (training)	
Input	Sequence of training examples $(\mathbf{x}_t, y_t) \in (\mathcal{R}^n, \{1, \dots K\})$, budget $\mathcal{B}$
Initialisation	For all $k \in \{1, \dots\}$ set $\mathbf{w}^{(k)} = 0$ ($\alpha_0^k = 0$), cache $\mathcal{S}^{(k)} = \varnothing$, distances $\mathcal{D}^{(k)} = \varnothing$, mistake count M = 0
Prediction	Compute the scores $s^{(k)}$ for each of the K classifiers: $s^{(k)} = \mathbf{w}^{(k)}(\mathbf{x}_t) = \sum_{i \in \mathcal{S}^{(k)}} \alpha_i^{(k)} K(\mathbf{x}_i, \mathbf{x}_t)$.
Update	For each classifier $\mathbf{w}^{(k)}$ for $k \in \{1, \dots, K\}$: 1. If the training label y_t is the same as the classifier label k: $y_t = k$, then set a new (temporary) label $\hat{y}_t = +1$, else set $\hat{y}_t = -1$. 2. If $\hat{y}_t * s^{(k)} \leqslant 0$ then we have misclassified for this classifier. Therefore: (a) $\mathcal{S}^{(k)} \leftarrow \mathcal{S}^{(k)} \cup \{\mathbf{x}_t\}$ (b) $\alpha_t^{(k)} \leftarrow \alpha_t^{(k)} - sign(s^{(k)})$ 3. While $\bigcup_{k=1}^{K} \lvert\mathcal{S}^k\rvert > \mathcal{B}$: (a) $\hat{i}, \hat{k} = \text{argmax}_{i,k \in \bigcup_{k=1}^{K} \mathcal{S}_i^{(k)}} \hat{y}_i(\mathbf{w}^{(k)}(\mathbf{x}_i) - \alpha_i^{(k)} K(\mathbf{x}_i, \mathbf{x}_i))$ (b) $\mathcal{S}^{(\hat{k})} \leftarrow \mathcal{S}^{(\hat{k})} - \mathcal{S}_{\hat{i}}(\hat{k})$ 4. $\mathbf{w}_{t+1}^{(k)}(\cdot) = \sum_{i \in \mathcal{S}^{(k)}} \alpha_i^{(k)} K(\mathbf{x}_i, \cdot)$ The final classification for the example $\mathbf{x}_t$ is the label $\hat{k}$ of the classifier $\mathbf{w}^{(k)}$ with the highest confidence score $\hat{k} = \text{argmax}_{k \in \{1,\dots,K\}} s^{(k)}$. If $\hat{k} \neq y_t$, then we have a mistake and we update the mistake counter $M = M + 1$

We then attempted to repeat parts a and b using the Forgetron algorithm. However, due to the shrinking and expanding Caches $\mathcal{S}^{(k)}$ we found it infeasible to employ vectorisation. This is where the theoretical and practical complexity depart. Although the major upside of Forgetron algorithm is that its time and memory complexity no longer scales with the number of training examples - as does the Perceptron's - the lack of vectorisation means that practically the algorithm is much slower with the digits dataset. We therefore used the suggested reduced dataset of 1000 training examples randomly sampled from the whole dataset (with an equal number of each digit) for the experiments.

Varying Budget $\mathcal{B}$: For our initial experiments we investigated how the performance of the algorithm varied with respect to the size of the allocated budget $\mathcal{B}$. Here for $d = 1, \dots, 7$ we trained and tested on the reduced data set with $\mathcal{B} = 200, 275, 350, 425, 500$, i.e. from 1/5 to 1/2 of the training set. The reported test error against budget size is reported below:

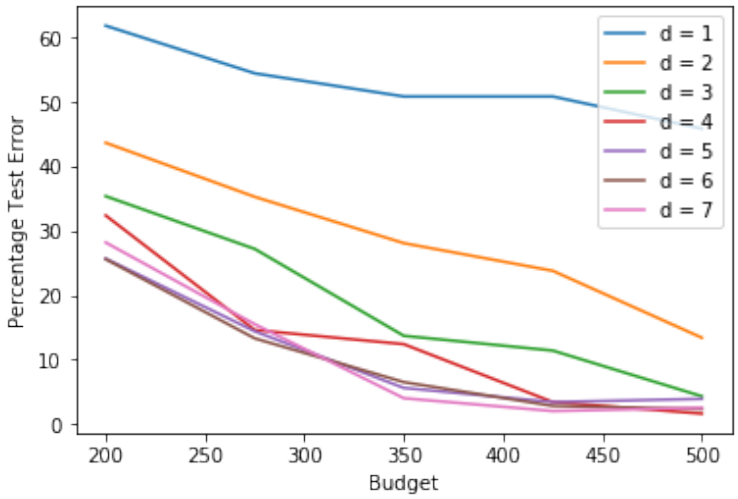


Figure 14: Test Error vs Budget

We observe that the for $d = 1, 2$ we note significantly poorer performance than the original Multi Class Perceptron, especially with a restricted budget. However, for $d = 5, 6, 7$ we get comparable performance with $\mathcal{B} = 350$ - a 1/3 of the original training set size.

Repeating part a: We now repeat the experiments of part a using a budget $\mathcal{B} = 350$. As noted above, this seems to give good performance but is also notably smaller than the size of the original training set, and so will provide give adequately interesting results with which to compare to the Perceptron. We now list the test errors for each d using 5 epochs and 20 runs. We also provide the statistics of the Perceptron algorithm for comparison:

	Training error	Test error
1	52.55% ± 2.95%	50.90% ± 10.10%
2	26.10% ± 1.28%	28.10% ± 6.32%
3	10.08% ± 1.68%	13.70% ± 1.60%
4	8.45% ± 0.83%	12.40% ± 2.08%
5	3.17% ± 1.92%	5.60% ± 1.80%
6	2.90% ± 0.72%	6.50% ± 2.10%
7	2.25% ± 0.57%	4.00% ± 0.95%

(a) Forgetron training and test error

	Training error	Test error
1	2.84% ± 0.50%	6.25% ± 1.98%
2	0.12% ± 0.12%	2.98% ± 1.03%
3	0.03% ± 0.05%	2.18% ± 1.13%
4	0.01% ± 0.04%	2.45% ± 1.15%
5	0.00% ± 0.00%	2.58% ± 1.15%
6	0.00% ± 0.00%	2.53% ± 1.36%
7	0.01% ± 0.03%	2.30% ± 0.90%

(b) Perceptron training and test error

Figure 15: Comparison of Forgetron, $\mathcal{B} = 350$ vs. Perceptron

Observe that despite using a budget of only 1/3 the size of the training data, the Forgetron test error for $d = 7$, (4.00%) is comparable to the test error for the Perceptron (2.30%). This suggests that despite its extremely poor performance for lower order polynomial kernels, there is some validity

to this approach.

1.8.2 Repeating part b

Using the same approach as for the Perceptron, we performed performed 5-fold cross validation in order to find the best d^*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
d*	7	6	6	7	6	6	7	7	6	7	6	6	6	7	7	7	6	7	7	7
test error	3.88%	6.31%	4.37%	2.43%	6.31%	6.80%	3.88%	9.22%	2.43%	1.46%	4.37%	6.80%	6.80%	8.74%	2.91%	3.88%	4.37%	4.85%	6.80%	6.31%

Figure 16: Forgetron cross validation

	Mean d* and std	Mean test error and std
1	6.55 ± 0.50	5.15% ± 2.06%

Figure 17: Forgetron best d^*

Observe that the mean test error for the mean d^* is now 5.15% with a standard deviation of 2.06%. This is 2.38% higher than the mean test error for the original Perceptron, with the standard deviation also 1.63% higher. This is to be expected - obviously discarding training examples from the final classifier will mean that the performance of the algorithm is worse. However, in a more elaborate example with a potentially much larger training set, this may be a trade-off that we have to make so that the computational complexity does not scale with the size of this set. Furthermore, the higher variance can be explained by the removal criteria. Upon performing multiple runs, the training examples removed from the cache $\mathcal{S}$ will be different and so the performance of the algorithm will be affected to varying degrees depending on the importance of these training examples for producing a 'good' classifier.

1.9 Support Vector Machines

As an alternative we chose to explore an algorithm widely used in regression and classification tasks, the Support Vector Machine (SVM). In the binary case, SVM is based on the idea that we wish to find the optimum way to separate the two classes by finding a separating hyperplane which lies as far away as possible from both classes. To do this, we focus on the points from each class which lie closer to the separating hyperplane, and hence influence the direction and position of the boundary the most. These points define a margin between the decision boundary and each class. Our goal is to maximise this margin and hence increase our confidence. The further apart the data points lie from the boundary the more certain we are that the data points are correctly classified. The SVM algorithm essentially maximises the distance between the separating hyperplanes and the points in the feature space closest to it. The necessary data structures *for the binary case* is as follows:

1. Kernel function $K : \mathcal{R}^n \times \mathcal{R}^n \to \mathcal{R}$
2. Input data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\} \in (\mathcal{R}^n, \{-1, 1\})^m$.
3. Regularisation parameter C which is a trade-off between the training error and the margin maximisation.
4. Parameters $\alpha_i, i = 1, \ldots, m$ where α solves the optimisation problem:

   $\boldsymbol{Maximise}\ -\frac{1}{2}\alpha^T A\alpha + \sum_i \alpha_i$

   $\boldsymbol{subject\ to}\ \sum_i y_i\alpha_i = 0\ \&\ 0 \leqslant \alpha_i \leqslant C\forall i$ where $A \in \mathcal{R}^{m\times m}, A_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$
5. Bias term b

Training an SVM is then equivalent to solving the optimisation problem in 4 above. Then the SVM function with Kernel K is defined as:

$$f(\mathbf{x}) = \sum_{i=1}^{m} y_i\alpha_i K(\mathbf{x}_i, \mathbf{x}) + b$$

and for a test point $\mathbf{x}^*$ we predict

$$y^* = sign(f(\mathbf{x}^*)).$$

Preliminary experimentation: We shall use the scikit-learn library `from sklearn.svm import SVC` in Python to implement the SVM algorithm for our data set. We will choose the *one vs one* method (as discussed in section 1.7) to generalise to K classes. A very important parameter of the SVM algorithm is the value of the parmeter C. As we mentioned above, C is a trade-off between the training error and the margin maximisation. This means that the smaller the value of C the more weight is given to the points far away from the boundary, and hence there is more tolerance towards misclassification. Similarly, as we increase the value of C we make our algorithm less tolerant towards misclassified points, hence focusing more on points closer to the decision boundary, while there is a bigger risk of over fitting to the training data if C gets too large. In this analysis we will cross-validate over different values of C ranging between C=0.1 and C=1000. Before performing cross-validation we should decide upon the kernel we will use. We perform a heuristic exploration of the Polynomial and the Gaussian kernels, both times using the same value for the parameter, C=1. The results are displayed below:

	Gamma value	Test error
1	0.00010	8.69% ± 0.61%
2	0.01675	2.47% ± 0.31%
3	0.03340	5.82% ± 0.50%
4	0.05005	17.82% ± 1.18%
5	0.06670	27.76% ± 1.20%
6	0.08335	48.70% ± 1.55%
7	0.10000	57.01% ± 1.14%

(a) SVM with Gaussian Kernel for C = 1

	d	Test error
1	1.0	4.61% ± 0.51%
2	2.0	3.15% ± 0.29%
3	3.0	2.79% ± 0.28%
4	4.0	2.98% ± 0.35%
5	5.0	3.51% ± 0.36%
6	6.0	4.85% ± 0.42%
7	7.0	7.15% ± 0.62%

(b) SVM with Polynomial Kernel for C = 1

Figure 18: Test errors for the SVM algorithm using the Gaussian and the Polynomial kernels

We observe that although the lowest test error for the Gaussian kernel (2.47%) is lower than the lowest test error for the Polynomial kernel (2.79%), the polynomial kernel performs significantly better in general. This however may be due to a poor choice for γ. Smaller values of γ correspond to smoother decision boundaries while larger values of γ correspond to a bigger influence of the decision boundaries by individual training points which can however lead to over fitting. We proceed by testing γ values closer to the best value obtained, namely 0.01675. The results for the range of γ from 0.01 to 0.02 are displayed below:

	Gamma value	Test error
1	0.010000	2.43% ± 0.32%
2	0.011667	2.26% ± 0.25%
3	0.013333	2.31% ± 0.25%
4	0.015000	2.64% ± 0.34%
5	0.016667	2.53% ± 0.32%
6	0.018333	2.64% ± 0.37%
7	0.020000	2.78% ± 0.42%

Figure 19: Test Error for the Gaussian Kernel SVM

We observe that for these values of γ, the Gaussian kernel achieves better test errors than the Polynomial kernel and will hence proceed using the Gaussian Kernel with a parameter value of 0.0117. We next consider the range of values of the parameter C we should experiment on. We first make some observations heuristically by trying different values of C and their effect on the performance of the algorithm. We observed that as the value of C increases the algorithm performs

better and that values of C smaller than 1 lead to significantly worse test errors. Thus, we perform 20 runs using 5 values of C equally spread in the range 1 to 10000.

Repeating part a The basic results are displayed below:

	C	Test error
1	1.0	2.45% ± 0.43%
2	10.0	2.20% ± 0.30%
3	100.0	2.21% ± 0.37%
4	1000.0	2.07% ± 0.23%
5	10000.0	2.17% ± 0.28%

Figure 20: Test Error for the Gaussian Kernel

Repeating part b The results obtained from cross validation of the values of C, using the Gaussian Kernel with parameter value 0.0117 in the SVM algorithm are displayed below:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C*	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
test error	2.31%	1.94%	2.15%	2.58%	2.42%	1.99%	2.10%	2.42%	2.10%	2.15%	2.63%	2.58%	1.67%	2.15%	1.56%	2.10%	2.10%	2.04%	1.61%	2.37%

Figure 21: Best value of C and mean test error

	Mean C* and std	Mean test error and std
1	10.00 ± 0.00	2.15% ± 0.30%

Figure 22: Mean value of C* and mean test error and standards deviation.

Even though the value of C = 100 gave the lowest test error in the basic results (Part a), after performing cross validation we observe that the value of C = 10 leads to the best test error in all of the 20 runs. This change in the optimal value of C could be related to the smaller size of the training set when performing cross validation. It is also worth noting that in order to improve the analysis of the SVM algorithm a search grid over both the kernel parameter and the parameter C could be made.

1.10 Comparison of performance of algorithms

1.10.1 Test Error

We compared the test error of each algorithm by taking the mean test error and standard deviation during the cross-validation phase

Comparison of Test Error		
Algorithm	Test Error (mean)	Standard Deviation
One vs All Perceptron	2.77% (Polynomial Kernel)	0.43%
One vs One Perceptron	3.35% (Polynomial Kernel)	0.35%
SVM	2.15% (RBF Kernel)	0.30%
Forgetron	5.15% (Polynomial Kernel)	2.06%

Comment out of all the algorithms, the SVM seems to perform the best, with the Forgetron (operating under a small budget) performing the worst. We would expect the SVM to be the best performing algorithm since it is not online and therefore reduces to a maximisation problem of finding the *best* separating hyperplane. Conversely, perceptron type algorithms which are online attempt to find *any* separating hyperplane as the data arrives sequentially.

As discussed before, with a limited budget the Forgetron will clearly have the worst performance and highest variance. However, an online algorithm whose time and space complexity scales with the number of training examples is not ideal and therefore a Forgetron is a possible solution to this problem.

The interesting distinction perhaps to be made is between the One vs All (OVA) and One vs One (OVO) Perceptrons. Both algorithms attempt to find separating hyper-planes but in slightly different ways. In a sense, the OVA algorithm holds a set of K perceptrons $\mathbf{w}^{(k)}$ for each class $k = \{1, \ldots, K\}$. Each perceptron is trained to distinguish whether a test point $\mathbf{x}^t$ is a member of its class or not - i.e. whether $\mathbf{x}^t \in k$ or $\mathbf{x}^t \notin k$. Conversely the OVO perceptron holds a set of $\frac{K(K-1)}{2}$ perceptrons each trained to distinguish whether a test point $\mathbf{x}^t$ has label i or j for all pairs of class labels (i, j) with $i, j \in \{1, \ldots, K\}$. The lower test error for the OVA perhaps suggests that the overall linear separability (in polynomial kernel space) between a *single* digit and *the rest* is more evident than between each *pair* of digits. Hence the better performance of the separating hyper-plane for the OVA algorithm.

1.10.2 Complexity

All the algorithms required a different number of epochs until the test error converged. For instance, the OVA required 5 epochs, whereas the OVO required 3 epochs. Therefore we timed runs of each algorithm for one cycle of training and testing until the test error converged.

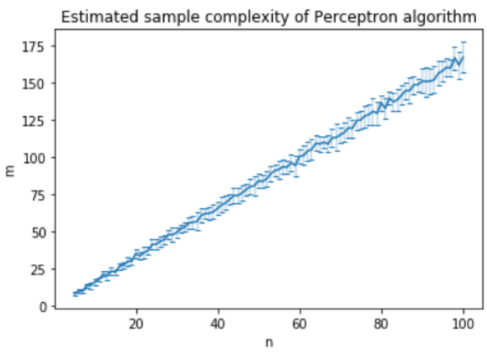
Comparison of Complexity		
Algorithm	Training and Test (seconds)	notes
One vs All Perceptron	10.23s	5 epochs
One vs One Perceptron	12.88s	3 epochs
SVM	6.49s	(library specific convergence criterion)
Forgetron	4.09s	5 epochs (limited data set, budget $\mathcal{B} = 350$)

Using the whole dataset the SVM is fastest overall. The OVA perceptron algorithm is slightly faster than the OVO likely as a result of using K perceptrons instead of $\frac{K(K-1)}{2}$. In order to keep results consistent we timed the Forgetron using the same limited dataset and budget as our previous Forgetron experiments. Clearly, the time complexity depends on the size of the training set and the budget allocated, however as noted before, without vectorisation no significant comparisons to the original Perceptron can be made. Perhaps an extension to the Forgetron to use vectorisation would allow for a more thorough analysis to be carried out.

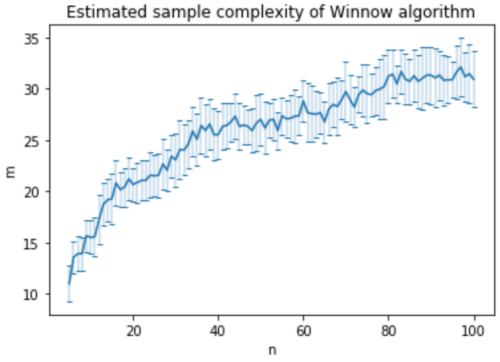
2 PART II

2.1 Part a

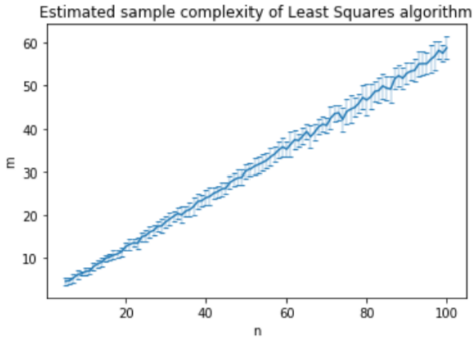
The plots of the estimated sample complexity for each of the four algorithms are displayed below:



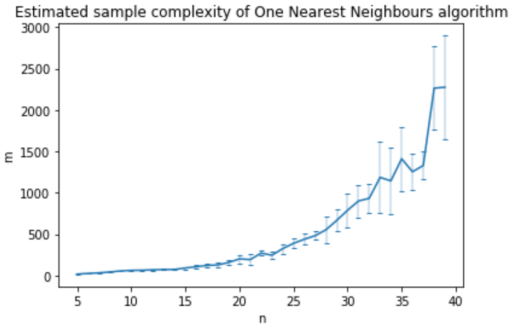
(a) Perceptron



(b) Winnow



(c) Least squares



(d) 1 Nearest Neighbour

Figure 23: Estimated sample complexity for the Perceptron, Winnow, Least Squares and 1 Nearest Neighbour algorithms.

2.2 Part b

The sample complexity estimations for each algorithm $\mathcal{A}$ required two approximations to be made. Firstly, the exact sample complexity requires the generalisation error defined to be:

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) \stackrel{def}{=} 2^{-n} \sum_{\mathbf{x} \in \{-1,1\}^n} \mathcal{I}[\mathcal{A}_{\mathcal{S}}(\mathbf{x}) \neq x_1] \tag{3}$$

However, computing the generalisation error over all possible $\boldsymbol{x} \in \{-1,1\}^n$ would be infeasible and therefore we calculated the empirical error over $N \leqslant 2^n$ randomly generated test points as a proxy for generalisation error:

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) \approx \mathcal{E}_{emp}(\mathcal{A}_{\mathcal{S}}) = \frac{1}{N} \sum_{\boldsymbol{x}^i : i=1,\ldots,N} \mathcal{I}[\mathcal{A}_{\mathcal{S}}(\boldsymbol{x}^i) \neq \boldsymbol{x}_1^i]$$

Over D random draws of a set $\mathcal{S}_m^{(d)}$ of m patterns drawn uniformly at random we therefore estimate the expected generalisation error as:

$$\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \approx \frac{1}{D}\sum_{d=1}^{D}\mathcal{E}_{emp}(\mathcal{A}_{\mathcal{S}_m^{(d)}}) \tag{4}$$

On a single run r_i we estimated the sample complexity as follows:

- for $\mathtt{n} = 5, \ldots, 100$:
    - set $\mathtt{m} = 1$
    - while $\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] > 0.1$
        - $\mathtt{m} = \mathtt{m} + 1$
        - calculate $\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \approx \frac{1}{D}\sum_{d=1}^{D}\mathcal{E}_{emp}(\mathcal{A}_{\mathcal{S}_m^{(d)}})$
    - estimated sample complexity on run r_i for dimension n is $\mathcal{C}_{r_i}(\mathcal{A}) = m$

The second approximation was that the sample complexity can be approximated as the average over R multiple runs, i.e

$$\mathcal{C}(\mathcal{A}) \approx \frac{1}{R}\sum_{i=1}^{R}\mathcal{C}_{r_i}(\mathcal{A}) \tag{5}$$

Specifically, for the Perceptron, Winnow and Least squares algorithms we had $N = 200$ test points $\mathbf{x}^i$ over $D = 10$ draws of training samples $\mathcal{S}_m^{(d)}$ with which to estimate generalisation error and performed $R = 20$ runs to estimate sample complexity. We thus had a trade off between accuracy and computational time by choosing the size of these three parameters. Within any one run, choosing a higher N and D will lead to a much more accurate estimate of expected generalisation error, however we need suitably small values of N, D for reasonable computational time. Furthermore we require enough runs R of the estimation procedure so that the variability of the estimates of $\mathcal{C}(\mathcal{A})$ decreases sufficiently and we have high enough confidence in the results (shown by the standard deviation bars). For the 1NN algorithm, the sample complexity grew much faster as a function of n, and so we only performed $N = 80$ test points from $D = 10$ draws over $R = 10$ runs. However, we found that our given parameters gave reasonable enough estimates for the general trend of the algorithms' sample complexities to become clear.

The bias of this method is that it systemically underestimates the sample complexity. This is a result of finding the *first* such m which has $\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})]$. Perhaps the estimation of m would be less optimistic if the minimum m that is found is then retested on larger range of test points, rather than the single set as in the algorithm

2.3 Part c

Experimentally, we can observe form the relevant graphs in part a that the sample complexity of the Perceptron and the Least squares algorithms grows linearly as a function of dimension n suggesting that $m = \Theta(n)$. However, we can see that although the sample complexity in both cases grows

linearly with n, in the case of the Perceptron, sample complexity is significantly larger than the one of the Least square's algorithm. In the Perceptron algorithm we observe that the estimated sample complexity m is of the order $m \approx 1.8 \times n$ whereas in Least squares it can be approximated from the graph that $m \approx 0.65 \times n$.

In the case of the Winnow algorithm, the graph suggests that the relationship between the estimated sample complexity m and the dimension of the data points n is of the form $m = \Theta(\log n)$. To visualise this we plot the estimated complexity of the Winnow algorithm together with three multiples of the the logarithm function. The graph obtained is displayed below:

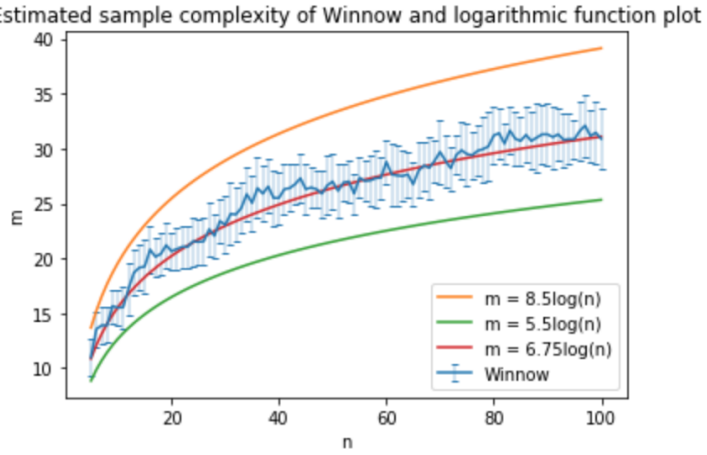


Figure 24: Plot of estimated sample complexity of the Winnow algorithm and multiples of the logarithmic function.

Based on our estimations, the sample complexity of the Winnow algorithm grows slower than the one of the other four algorithms and achieves the best sample complexity for large values of n. It is therefore a significantly better choice, in terms of sample complexity, in cases where we would like to classify data points in very large dimensions.

Finally, based on the graph 1NN has sample complexity upper bounded by $O(2^n)$ and lower bounded by $\Omega(n^2)$. However it is hard to ascertain tighter bounds just based on the graph.

2.4 Part d

In lecture we saw the Perceptron bound Theorem by Novikoff which states that for all sequences of examples $S = (x_1, y_1), \ldots, (x_m, y_m) \in \mathbb{R}^n \times \{-1, 1\}$ the mistakes of the Perceptron algorithm is bounded by $M \leqslant (\frac{R}{\gamma})^2$ whith $R = \max_t ||x_t||$, if there exists a vecotr $\boldsymbol{v}$ with $||\boldsymbol{v}|| = 1$ and a constant γ such that $(\boldsymbol{v} \cdot \boldsymbol{x}_t) y_t \geqslant \gamma$. In our case, we have $S = (x_1, y_1), \ldots, (x_m, y_m) \in \{-1, 1\}^n \times \{-1, 1\}$ and hence

$$||\boldsymbol{x}_t|| = \sqrt{n} \text{ for all } t = 1, \ldots, m.$$

Choose $\boldsymbol{v} = (1, 0, \ldots, 0)$, then $||\boldsymbol{v}|| = 1$ and for any $t \in \{1, \ldots, m\}$, we have that

$$y_t(\boldsymbol{x}_t \cdot \boldsymbol{v}) = \boldsymbol{x}_{t_1} y_t = \boldsymbol{x}_{t_1}^2 = 1$$

where $\boldsymbol{x_{t_1}}$ denotes the first entry of $\boldsymbol{x}_t$ which is exactly equal to y_t in this problem. Hence we can choose $R = \sqrt{n}$ and $\gamma = 1$ in the above theorem and we have that the number of mistakes by the perceptron algorithm are bounded above by n.

We also proved in lectures that given m and letting t be drawn uniformly at random from $\{1, \ldots, m\}$, we have

$$P(\mathcal{A}_{\mathcal{S}_t}(\boldsymbol{x}_{t+1}) \neq y_{t+1}) \leqslant \frac{B}{m}$$

where B is a mistake bound for algorithm $\mathcal{A}$. In other words, using the mistake bound derived above, we have

$$\hat{p}_{m,n} \leqslant \frac{n}{m}.$$

2.5 Part e