Supervised Learning (COMP0078) - Coursework 2

Team: NO, Student number: 21146187

December 20, 2021

1. Kernel perceptron

Generalizing the Kernel perceptron to k classes

Method 1

For binary classification with N training points, we predict the class $\hat{y}$ of a data point $\mathbf{x}$ according to

$$\hat{y} = \text{sign}\left(\sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i K(\mathbf{x}_i, \mathbf{x})\right) \tag{1}$$

where $\mathcal{D}$ is the set of training data, α_i are the set of learned weights and K is the kernel function. This can be extended to multi-class classification involving k classes, where instead the classification output will be a vector $\boldsymbol{\kappa} \in \mathbb{R}^k$, whose ith element represents the confidence of being in the ith class. We can write

$$\kappa_a = \sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i^{(a)} K(\mathbf{x}_i, \mathbf{x}) \tag{2}$$

where $\boldsymbol{\alpha}^{(a)}$ is the ath set of learned weights represented as a vector (this corresponds to the set of weights for class a). We may write this in a vectorized format:

$$\boldsymbol{\kappa} = \alpha \boldsymbol{K}_{\mathbf{x}} \tag{3}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^{k \times N}$ is the matrix formed by row stacking the $\boldsymbol{\alpha}^{(a)}$ $\forall a \in \{1, \ldots, k\}$ vectors, and $\boldsymbol{K}_{\mathbf{x}} \in \mathbb{R}^N$ is the vector whose ith element is $K(\mathbf{x}_i, \mathbf{x})$. The class $\hat{y}$ of $\mathbf{x}$ is then predicted with the maximum index of $\boldsymbol{\kappa}$ ($\underset{a}{\text{argmax}}(\kappa_a)$).

We define a misclassification of point $\mathbf{x}_t$ as when $\underset{a}{\text{argmax}}(\kappa_a) \neq y_t$. We update the parameters (elements of $\boldsymbol{\alpha}$) during training as follows: For a training point $\mathbf{x}_t$, whose label is $y_t = c$ (where c is one of the class labels), if there is no misclassification then nothing is changed. If there is a classification error (i.e if there exists any b for which $\kappa_b \geq \kappa_c$, then the updates are

$$\alpha_i^{(c)} \to \alpha_i^{(c)} + K(\mathbf{x}_i, \mathbf{x}_t) \tag{4}$$

$$\alpha_i^{(b)} \to \alpha_i^{(b)} - K(\mathbf{x}_i, \mathbf{x}_t) \tag{5}$$

i.e in an instance of misclassification, the parameters of the correct class are increased, the parameters of any classes whose confidence is higher than (or equal to) the confidence of the correct class are decreased and the parameters of any other classes remain unchanged. The matrix $\boldsymbol{\alpha}$ is initialised with all its elements set to 0. The training data is looped through

	k-class Kernel Perceptron (training) method 1
Input	$\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n, \{0, \dots, k-1\})^m$
Initialization	$\alpha_i^{(a)} = 0 \ \forall a \in \{0, \dots, k-1\}, \forall i \in \{1, \dots, m\}$
Prediction	Upon receiving the tth instance $\mathbf{x}_t$, predict $\hat{y}_t = \underset{a}{\operatorname{argmax}}(\kappa_a) = \underset{a}{\operatorname{argmax}} \left(\sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i^{(a)} K(\mathbf{x}_i, \mathbf{x}_t) \right)$
Update	(define $\mathbf{x}_t \in$ class c, (i.e $y_t = c$)) `if` $\hat{y}_t = y_t$ `then` do nothing `else` $\alpha_i^{(c)} \to \alpha_i^{(c)} + K(\mathbf{x}_i, \mathbf{x}_t)$ $\alpha_i^{(b)} \to \alpha_i^{(b)} - K(\mathbf{x}_i, \mathbf{x}_t)$ $\{b \vert \kappa_b \geq \kappa_c\}$, $i \in \{1, \dots, m\}$
	loop through all training data

Implementation: (How $\sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i^{(a)} K(\mathbf{x}_i, \cdot)$ was i) represented, ii) evaluated and iii) how new terms are added to the sum during training)

Let N be the number of training points. The kernel matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ was first created by taking the scalar product between all data points of the training data. The matrix $\boldsymbol{\alpha} \in \mathbb{R}^{k \times N}$, was initialised as described in the previous section. The classification of the ith training point is evaluated by performing the matrix multiplication of $\boldsymbol{\alpha}$ and the ith column of $\mathbf{K}$. New terms are not 'added to the sum during training', as we are dealing with the entire $\mathbf{K}$ and $\boldsymbol{\alpha}$ matrices from the outset. Instead, the columns of $\mathbf{K}$ are looped over and $\boldsymbol{\alpha}$ is correspondingly updated during training.

Method 2

This method instead uses k binary classifiers to specify whether a data point belongs to class c or doesn't belong to class c, $\forall c \in \{1, \ldots, k\}$. In the previous method, for some data point $\mathbf{x}_t$, its corresponding class label y_t was an element from $\{0, \ldots, k-1\}$. For method 2, we recast all the labels of the data into k-dimensional vectors; $y_t \rightarrow \mathbf{y}_t$, where all the elements of $\mathbf{y}_t$ are -1, except for its y_tth element, which is $+1$. For example, in a 5 class classification problem, if some $y_t = 1$, then $\mathbf{y}_t = (-1, +1, -1, -1, -1)$. Method 2 follows a similar formulation to Method 1, but in this case

$$\kappa_a = \text{sign}\left(\sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i^{(a)} y_i^{(a)} K(\mathbf{x}_i, \mathbf{x}) \right) \tag{6}$$

Where $y_i^{(a)}$ is the ath element of $\mathbf{y}_i$. The resulting vector $\boldsymbol{\kappa}$ will have all elements $\in \{-1, 0, 1\}$, and the classification is chosen at random from the indices of $\boldsymbol{\kappa}$ whose elements are $+1$, or in the case where no elements are $+1$, the class is chosen at random from all indexes whose elements are 0, or in the case where all elements are -1, the class is chosen at random from all indices. In the training step the overall classification does not actually contribute to updating of parameters, only the classification of the k individual binary classifiers matter.

	k-class Kernel Perceptron (training) method 2
Input	$\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), \dots (\mathbf{x}_m, \mathbf{y}_m)\} \in (\mathbb{R}^n, \mathbb{R}^k)^m$
Initialization	$\alpha_i^{(a)} = 0 \ \forall a \in \{0, \dots, k-1\}, \forall i \in \{1, \dots, m\}$
Prediction	Upon receiving the tth instance $\mathbf{x}_t$, $\forall a$ predict $\hat{y}_t^{(a)} = \kappa_a = \text{sign}\left(\sum_{\mathbf{x}_i \in \mathcal{D}} \alpha_i^{(a)} y_i^{(a)} K(\mathbf{x}_i, \mathbf{x}_t)\right)$
Update	$\forall a$ `if` $\hat{y}_t^{(a)} = y_t^{(a)}$ `then` do nothing `else` $\alpha_t^{(a)} \to \alpha_t^{(a)} + 1$
	loop through all training data

Results

(For reference the polynomial kernel is $K_d(\mathbf{p}, \mathbf{q}) = (\mathbf{p} \cdot \mathbf{q})^d$ and the Gaussian kernel is $K_c(\mathbf{p}, \mathbf{q}) = \exp(-c\|\mathbf{p} - \mathbf{q}\|_2^2)$)

Table of means

For different kernels and different methods, 20 runs were performed for $d = 1, \dots, 7/c = 0.5, 1, 1.5, \dots, 3.5$, with each run randomly splitting the total data into a 4:1 training:test split. In all three cases the number of epochs for training is 2 (this is due to time constraints). The mean test and training errors for each d are presented below. The range of values of c was decided by performing 3 runs each for values between 0 and 10 (using the smaller test and train data), and roughly viewing which range had the lowest error.

Polynomial Kernel (method 1)

.

	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
Train error (%)	27.3 ± 9.6	17.8 ± 4.2	15.4 ± 3.2	11.9 ± 2.8	11.6 ± 2.6	10.7 ± 4.1	9.3 ± 2.0
Test error (%)	27.7 ± 9.7	18.3 ± 4.6	15.8 ± 3.7	12.8 ± 3.0	12.6 ± 2.6	12.0 ± 4.1	11.1 ± 2.0

Gaussian kernel (method 1)

.

	$c = 0.5$	$c = 1$	$c = 1.5$	$c = 2$	$c = 2.5$	$c = 3$	$c = 3.5$
Train error (%)	0.05 ± 0.06	0.001 ± 0.006	0.003 ± 0.007	0	0	0	0
Test error (%)	5.0 ± 1.3	5.3 ± 1.3	4.6 ± 0.5	5.1 ± 0.5	5.3 ± 0.5	5.5 ± 0.6	5.8 ± 0.5

Polynomial kernel (method 2)

.

	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
Train error (%)	8.2 ± 8.5	4.3 ± 4.6	2.5 ± 2.7	2.2 ± 2.3	1.8 ± 1.9	1.7 ± 1.9	1.6 ± 1.6
Test error (%)	8.9 ± 9.2	5.6 ± 5.9	4.1 ± 4.3	3.8 ± 4.0	3.4 ± 3.5	3.4 ± 3.6	3.2 ± 3.3

Cross-validated optimal parameter

In each three cases cross-validation is performed (as described in the coursework 2 question sheet). This is used to find an optimal parameter (d^*/c^*), which is then used to retrain and test on a random 4:1 split of the entire data set. This is performed 20 times to obtain mean parameters and test errors along with their standard deviations. The results are presented below.

Polynomial kernel (method 1)

	d^*	Test Error (%)
Mean	4.9	11.55
standard deviation	0.9	2.47

Gaussian Kernel (method 1)

	c^*	Test Error (%)
Mean	2.3	6.1
standard deviation	0.6	0.6

Polynomial Kernel (method 2)

.

	d^*	Test Error (%)
Mean	4.9	7.1
standard deviation	1.1	1.1

Confusion matrix

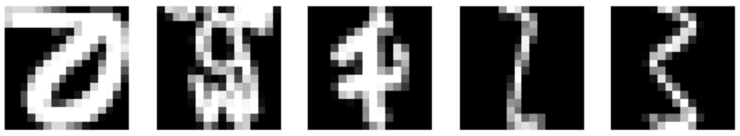
Cross validation is performed to select an optimal d in the same way as the previous section. This optimal d is then retrained on the full 80% training set and tested on the remaining data to produce a confusion matrix; the rows correspond to the actual classes of the test data (i.e 1,...,9), the columns correspond to what the data was WRONGLY predicted. This was done 20 times to obtain a mean matrix with a standard deviation. The results are presented below.

Polynomial Kernel (method 1): Confusion Matrix

real	Predicted digit									
digit	0	1	2	3	4	5	6	7	8	9
0	0	0.2±0.4	2.7±2.8	1.5±2.3	1.3±1.0	1.7±2.9	3.4±3.7	0.7±0.9	0.6±0.9	0.4±0.8
1	0	0	0	1.0±2.1	0.9±1.6	0.2±0.4	0.6±0.7	0.5±1.4	1.5±2.1	0.6±1.1
2	2.1±2.7	2.1±2.8	0	6.7±10.2	5.1±6.0	0.6±0.9	2.2±3.0	5.3±6.6	3.5±2.6	1.6±2.0
3	1.1±1.5	0.2±0.4	2.9±3.2	0	0.6±0.6	4.9±4.2	0	2.5±4.2	4.4±4.7	2.2±3.8
4	0.3±0.9	7.4±5.3	4.9±3.4	1.4±4.2	0	0.2±0.5	2.8±2.1	2.5±4.3	4.1±6.6	12.3±10.7
5	2.4±2.1	0.7±1.3	1.5±1.6	10.4±11.8	4.2±3.3	0	3.1±3.6	1.3±3.1	4.4±7.8	3.7±3.3
6	2.7±1.9	2.0 ±1.9	2.3 ±2.0	0.8±2.6	1.4±1.8	1.6±2.7	0	0.5 ±1.1	1.9±2.7	0.2±0.5
7	0.2±0.5	0.7 ±0.9	1.0 ±1.9	2.7 ±8.9	2.3 ±2.2	0.3±0.7	0.2 ±0.5	0	1.5±1.3	13.0±15.3
8	1.8 ±2.2	2.4±2.9	3.0±2.2	8.7 ±8.3	3.3±2.9	2.8±2.4	0.8±1.5	1.4 ±2.6	0	4.3±5.8
9	0.3±0.4	1.1±1.2	0.3±0.6	3.0±6.8	4.4±5.8	0.4±0.9	0	8.1±14.4	1.8±2.7	0

The 5 hardest digits to predict (in some random 20% test set) for polynomial kernel (method 1))

The method used to decide the most difficult digits was to normalise all the $\boldsymbol{\kappa}$ vectors of the wrongly predicted digits, then choose the 5 digits whose $\boldsymbol{\kappa}$ vectors' elements corresponding to the correct class were most negative. This essentially chooses the 5 digits whose confidences of their correct class were smallest. The digits with their correct labels are shown below. It is understandable how these are the most difficult to predict, as they are stretched/squashed/rotated/have extra lines/have missing lines.



```
tensor([0., 9., 7., 2., 3.])
```

Figure 1: most difficult to predict digits (in some random 20% test set)

Comparison of polynomial and Gaussian kernels

In the situation that has been tested hear, it is clear that the Gaussian kernel is superior (in terms of smaller generalisation error). The test error of the Gaussian also had a much smaller proportional standard deviation than the polynomial kernel, this suggests that the Gaussian kernel converges quicker during training. However, it is difficult to definitively say which is better, as only 2 training epochs have been used in this case.

2. Spectral Clustering

Overview of Implementation:

1. A set of data points X, I constructed the weight matrix W by finding the squared 2-norm between all the points, multiplying by $-c$ (the Gaussian scale factor), and then exponentiating.

2. I constructed the degree matrix D by summing over the rows/columns and recasting the resulting vector as a square matrix. I found the Laplacian L by taking $D - W$

3. I found $\mathbf{v}$, the eigenvector of L corresponding to its second smallest eigenvalue (by using a pytorch in-built function).

4. I found the predictions of clusters of by taking the sign of $\mathbf{v}$, and setting all elements of 0 to $+1$. This produced a vector containing the predicted clusters of each data point.

The clustering algorithm was performed on the two-moons dataset, the results are shown in figure 2. A value of c which correctly clusters the data is $c = 16$.

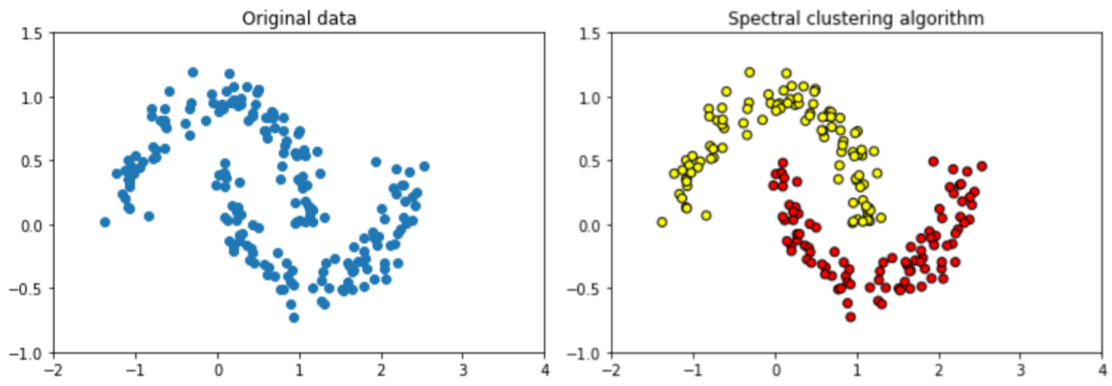


Figure 2: Two-moons data (left), correctly clustered data using spectral clustering with $c = 16$ (right)

The clustering algorithm was performed on synthetic data sampled from 2 separate Gaussians. The results are shown in in figure 3. A value of c which correctly clusters the data is $c = 2$.

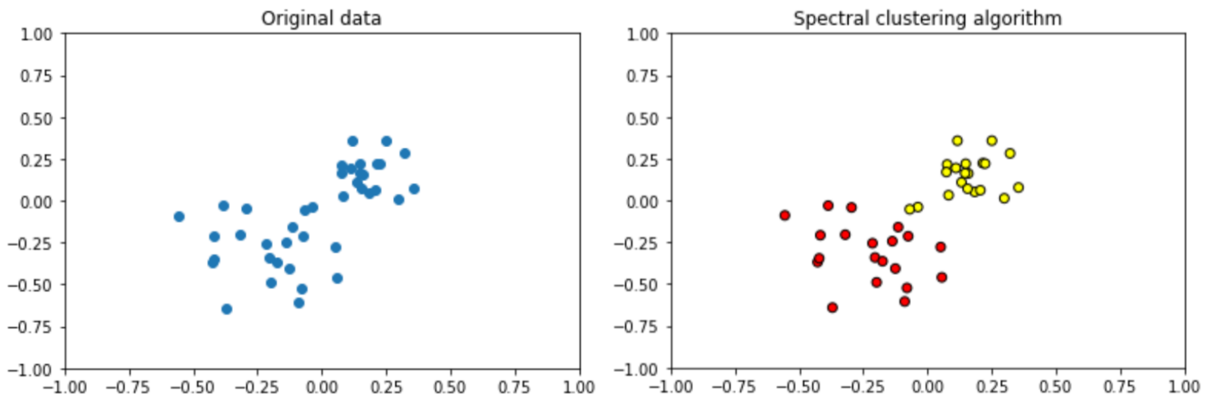


Figure 3: Synthetic data sampled from two separate Gaussians (left), correctly clustered data using spectral clustering with $c = 2$ (right)

The clustering algorithm was performed on real data (MNIST data consisting of only the digits '1' and '3'. The fraction of correctly clustered points is plotted against varying c in figure 4. The value of c that performs best inside the range [0,0.5] is $c = 0.319$ (to the nearest 1000th decimal).

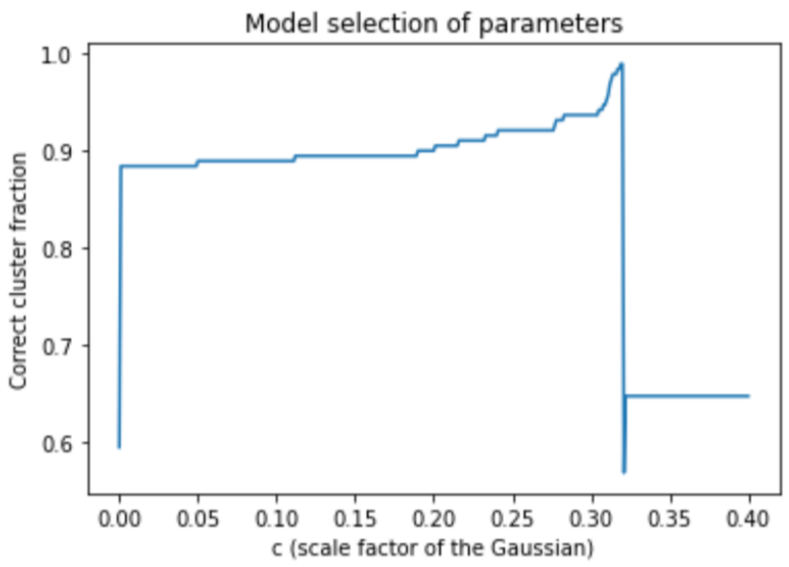


Figure 4: Correct cluster fraction plotted against c for binary clustering of MNIST digits

Questions

1. $\mathrm{CP}(c) := \frac{\max\{l_-, l_+\}}{l}$. l is the number of data points, l_+ is the number of points that have been assigned the correct label, l_- is the number of points that have been assigned the incorrect label. $\mathrm{CP}(c)$ is a reasonable measure of cluster correctness as we don not care about the correctness of the labels, just how accurate the clusters are, i.e the points could be clustered correctly but the labels of the two clusters could be swapped around from the true labels. If the points have been clustered, but all labelled incorrectly then the true labels of the points have just been swapped from -1 to 1 and vice versa. This means they are 100% correctly clustered. In this case $l_-/l = 1$, which correctly reflects this.

2. The smallest eigenvalue of the Laplacian and its corresponding eigenvector. We first show that the Laplacian is positive semi-definite.

$$\begin{aligned}
\mathbf{u}^\intercal L\mathbf{u} &= \mathbf{u}^\intercal (D - W)\mathbf{u} && (7)\\
&= \sum_{i,j} D_{ij}u_iu_j - \sum_{i,j} W_{ij}u_iu_j && (8)\\
&= \sum_i D_{ii}u_i^2 - 2\sum_{i<j} W_{ij}u_iu_j && (9)\\
&= \sum_i \left(\sum_j W_{ij}\right) u_i^2 - 2\sum_{i<j} W_{ij}u_iu_j && (10)\\
&= \sum_{i<j} W_{ij}(u_i^2 + u_j^2) - 2\sum_{i<j} W_{ij}u_iu_j && (11)\\
&= \sum_{i<j} W_{ij}(u_i - u_j)^2 && (12)\\
&= \sum_{i<j} e^{(-c\|\mathbf{x}_i - \mathbf{x}_j\|^2)}(u_i - u_j)^2 && (13)\\
&\geq 0 && (14)
\end{aligned}$$

Let $\mathbf{v} \neq \mathbf{0}$ be an eigenvector of L with corresponding eigenvalue λ. Then from above:

$$\begin{aligned}
\mathbf{v}^\intercal L\mathbf{v} &\geq 0\\
\lambda \mathbf{v}^\intercal \mathbf{v} &\geq 0\\
\lambda \|\mathbf{v}\|^2 &\geq 0\\
\lambda &\geq 0
\end{aligned}$$

Hence, if any eigenvectors with an eigenvalue of 0 exist, then 0 will be the smallest eigenvalue. From our proof of positive semi-definiteness, we know that

$$\mathbf{v}^\intercal L\mathbf{v} = \sum_{i<j} e^{(-c\|\mathbf{x}_i - \mathbf{x}_j\|^2)}(v_i - v_j)^2$$

It is trivial to see that this equals 0 if and only if $\mathbf{v}$ is the constant vector. Hence 0 is the smallest eigenvalue with corresponding eigenvalue being the constant vector.

3. Why spectral clustering works: If we partition a graph by assigning each vertex a value $x = \pm 1$. For a pair of vertices v_i and v_j belonging to different clusters, $(x_i - x_j)^2 = 4$ (this is the only possible value. Obviously, if v_i and v_j belong to the same cluster, $(x_i - x_j)^2 = 0$ So the number of edges between partitions is proportional to

$$\sum_{i,j} (x_i - x_j)^2$$

The total weight of the edges between the partitions is then proportional to

$$\sum_{i,j} e^{(-c\|\mathbf{x}_i - \mathbf{x}_j\|^2)} (x_i - x_j)^2 = \mathbf{x}^\intercal L \mathbf{x} \tag{15}$$

This quantity should be small for a good partition (as in general, intra-cluster edges should have large weights, and inter-cluster edges should have small weights). Minimising the above with the constraint $\mathbf{v}^\intercal \mathbf{v} = |V|$ (where $|V|$ is the number of nodes) gives

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

where λ is the Lagrange factor, which turns out to be an eigenvalue. Clearly the eigenvector corresponding to the 0 eigenvalue doesn't work, as it assigns all vertices to the same cluster. The intuition is that the smaller the eigenvalue, the smaller the total weight of the edges between the two clusters. So we choose the eigenvector corresponding to the 2nd smallest eigenvalue. However, the elements of this may not be ± 1, so we take the sign of this vector.

Reference

4. How c influences the quality of clustering: Large c means the weight decays

faster between points that are physically farther from each other. A small c (near 0) assigns a similar weight (1) to all edges between points, regardless of distance from each other. $c \simeq 0$ means the Laplacian would be approximate to a matrix of all ones, this would result in any arbitrary EQUAL SIZED clusters being equally likely, as this would minimize the equation 15 (i.e useless clustering). A large c corresponds to $L \simeq$ Identity matrix. Meaning $\mathbf{x}^\intercal L\mathbf{x} \simeq \mathbf{x}^\intercal\mathbf{x} = |V|$. This means any VERY IMBALANCED sized clusters is equally likely. Clearly there needs to be a balance between small and large c for good clustering quality.

3. Sparse Learning

(a)

The plots are shown in Figure 5.

(b)

Algorithm classes were defined for each of the algorithms, each class took as input m and n, (the training sample size the and dimension respectively). The training sample was generated as a class attribute (i.e there is a 1 unique training sample for any instantiation of a class).

For the WINNOW and perceptron classes, there was a method to train using the training data and some set number of epochs. Each class also had a method to output the error rate of predictions on an arbitrary test set.

(i)

1. For each n, test data consisting of N samples was generated.

2. Starting from $m = 1$, the algorithm class was instantiated, and the error rate on the test data was found. If the error rate was above 0.1, m was incremented. This a repeated until the error rate is below 0.1, then m is recorded.

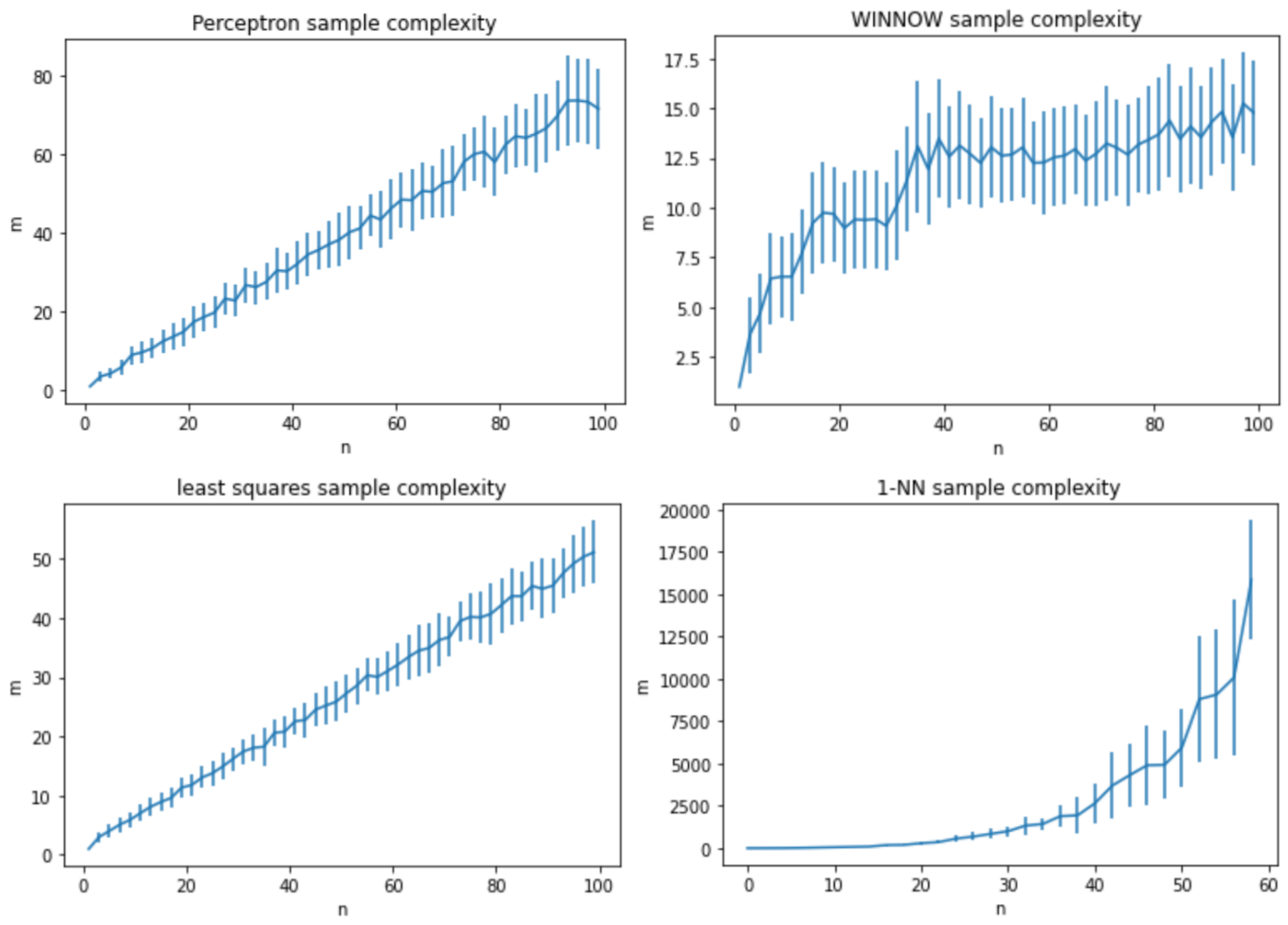


Figure 5: Sample complexity plots for Perceptron, winnow, least squares and 1-NN

3. Step 2 is repeated C number of times to obtain multiple values of m, this is used to calculate a mean and standard deviation.

For least-squares, perceptron and 1-NN, the values of N and C were 100 and 50 respectively. The perceptron was trained 20 times for each increment in step (2). For WINNOW, N and C were 200 and 150 respectively (this was due to the error bars being proportionally much larger than in the other 3 cases).

(ii)

Tradeoffs and biases of method: A trivial tradeoff is time for accuracy, as having a finite number of test samples gives an estimation of the generalisation error, but also makes the implementation computationally tractable.

The most accurate method would be to train the classifiers on multiple sets of training data for each m, and then find an average test error for each m, then increment m if this value is below 0.1. I instead only trained on 1 set of training data for each m, this trades accuracy for time. However, a new set of training data was created and trained on multiple times to produce an estimate of the mean m, using multiple sets of training data, biases inherent in a single set of data are averaged out.

(c)

From figure 5, my estimate of the sample complexities of the perceptron, winnow, least squares and 1-NN is $m = \Theta(n)$, $\Theta(\log n)$, $\Theta(n)$, and $\Theta(e^n)$ respectively. Although both the perceptron and the least squares have sample complexities that scale linearly, the plot for the perceptron has a higher gradient, showing that the least squares performs slightly better in a sparse data situation. The order of the algorithms by performance (at the edge of the sparse regime) from best to worst is: Winnow, least-squares, perceptron, 1-NN.

(d)

Upper bound on probability of mistake on sth example after perceptron has been trained on $s-1$ examples where $s \in \{1, \ldots, m\}$ is sampled uniformly at random:

If the upper bound on the number of mistakes for a set of examples of size m is B, then there are no more than B trials with mistakes, since s is drawn uniformly from $\{1, \ldots, m\}$ there is no more than a B/m probability of making a mistake at the sth trial.

B is give by the Novikoff Perceptron Bound, (no. of mistakes $M \leq B = (R/\gamma)^2$, where R is the maximum magnitude of any data point and γ is the separation margin of the data. From the construction of the data $\mathbf{x} \in \{-1, 1\}^n$, we know that $\|\mathbf{x}\| = \sqrt{n} \implies R = \sqrt{n}$. From the labelling of the data ($y = x_0$), we see thay

$\gamma = 1$. Hence:

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