Coursework 2

COMP0078: Supervised Learning

SID: 25226382 and 25150451

Due: 5th January 2026

Part I

1. The One-versus-rest strategy works as follows: We first train k seperate binary classifiers (one for each class), for the i'th classifier we relabel data as $y = +1$ if the true class is i and y=-1 otherwise so the classifier learns to distinguish class i from the rest. For a given test point x we evaluate all classifiers to get their scores:
$$f_i(x) = \sum_{t=1}^{N} \alpha_t^{(i)} K\big(x^{(t)}, x\big)$$
We then finally predict:
$$\hat{y} = \arg \max_{i \in \{1,\ldots,k\}} f_i(x)$$
This results in a simple algorithm where each classifier is slightly imbalanced as it is doing 1 class vs k-1 classes.

2. The code implementation is in the coding zip file. The implementation trains k binary kernel perceptrons where classifier i uses labels +1 for class c and -1 otherwise. At test time we compute and predict scores based on the above algorithm.

How $\mathbf{w}(\cdot)$ is represented

The weight vector $\mathbf{w}$ is not stored explicitly as it lies in a potentially infinite dimensional feature space. Instead, it is represented implicitly by:
$$\mathbf{w} = \sum_{i \in \text{AVs}} \alpha_i \phi(\mathbf{x}_i)$$
where AV is the set is active indices. In our implementation, active_indices stores the indices of the training samples which caused errors where i_j = active_indices[j] and alpha stores the corresponding contributions where $\alpha_j = alpha[j]$ and $\alpha_j \in \pm 1$. This implicitly represents w without ever computing it by way of kernel methods.

Evaluating the sum

To compute $\mathbf{w} \cdot \phi(\mathbf{x})$ for a new input:
$$f(\mathbf{x}) = \sum_{i \in \text{AVs}} \alpha_i K(\mathbf{x}_i, \mathbf{x})$$
We can then of course compute this by evaluating the polynomial or gaussian kernel at the given points.

How new terms are added to the sum during training

When a mistake is made on example $\mathbf{x}_t$ with label y_t: We update the accumulator:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_t\phi(\mathbf{x}_t)$$

and append index t to `active_indices` and y_t to `alpha`.

Epochs

We chose 5 epochs as this balances training time with ability to converge on seperable data. This was chosen heuristically but proved to be a good choice empirically.

3. Q3 Results (Mean Error ± Std Dev)

	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
Train	0.0778 ± 0.0415	0.0115 ± 0.0170	0.0032 ± 0.0057	0.0017 ± 0.0043	0.0004 ± 0.0002	0.0003 ± 0.0001	0.0003 ± 0.0001
Test	0.0967 ± 0.0380	0.0406 ± 0.0154	0.0312 ± 0.0049	0.0289 ± 0.0054	0.0286 ± 0.0037	0.0273 ± 0.0032	0.0281 ± 0.0030

We can see a clear trend in the training error which decreases monotonically as polynomial degree increases and stagnates at $d \geq 6$. The higher degree polynomials logically have more degrees of freedom to fit the data and end up overfitting. Test error initially decreases sharply (d=1 to d=2: 9.7% -> 4.06%) and continues improving gradually until reaching a minimum at d=6 (2.7%). After the 6th degree polynomial, test error increases. This shows demonstrates the tradeoff that we can expect from higher polynomial degrees. THe highest dimension model has a very low training error but unimproved test error which indicates that it is overfitting the training data rather then developing the ability to generalize. The dimensions around 4-7 seem to be the sweet spot for this experiment as that is where we see test error start to stagnate, lower has poor training error and higher would likely see increased test error.

4. Results:

- Mean d*: 5.12 ± 1.06
- Mean Training Error: 0.0006 ± 0.0004
- Mean Test Error: 0.0282 ± 0.0028

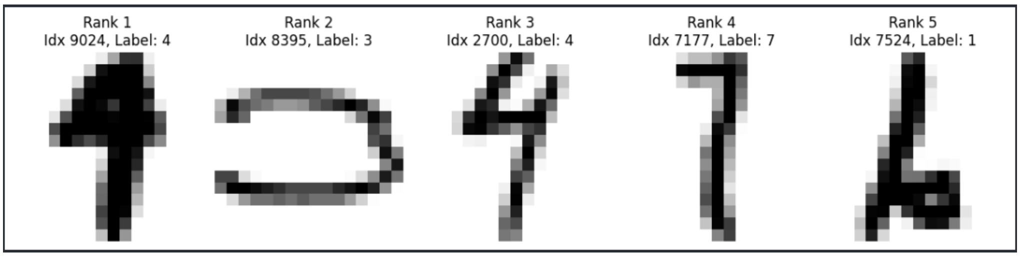
Cross validation most often selects dimensions between 4 and 6. This aligns with the findings from q3. Our mean d^* of 5.15 and standard deviation of only 1.06 tells us that cross validation is consistently finding a similar optimal range. The CV test indicates to us that the models with dimension ≥ 4 have sufficient capacity to separate the data as no model with a degree lower than 4 was ever selected. The low standard deviation of 1.06 indicates to us that random splits lead to similar conclusions about the most optimal degree, our CV approach is robust based on this.

5. Confusion matrix is below. The most commonly confused digits are 8->3, 3 -> 5, 5 -> 3, 8 -> 5, 9 -> 4 and 9 -> 7 (true -> pred). These results make intuitive sense as 3 and 5 share very similar curved features. A 5 with a curved bottom can look exactly like a 3. This lead to 3 -> 5 having the highest error at 1.7%. Similarly, 8 has loops that can resemble certain parts of 5. 9 and 4 are very very similar visually as well which is shown in the results table and in the following question. The most well separated digits include 0, 1 and 7. These all have very low confusion rates which is also quite intuitive I think. All of these digits have distinct shapes, 0 is a oval loop, 1 is a simple vertical line and 7 unique angled lines. One interesting observation that we can make from the confusion matrix is that the confusions are not symmetric, that is, 3 -> 5 gets confused more then 5 -> 3 which shows the variation between data points in the same class. In terms of the overall hardest digits

to classify, 8 and 5 have the highest total row error. The confusion matrix shows that errors are concentrated among digits with similar visual features (loops and curves in particular) while unique digits are easily classified which indicates that our model is working as expected.

True \ Pred	0	1	2	3	4	5	6	7	8	9
0	0 ± 0	0.001 ± 0.002	0.001 ± 0.002	0.001 ± 0.002	0.000 ± 0.001	0.002 ± 0.003	0.002 ± 0.002	0.001 ± 0.002	0.001 ± 0.002	0.001 ± 0.001
1	0.000 ± 0.001	0 ± 0	0.001 ± 0.001	0.000 ± 0.001	0.002 ± 0.003	0.000 ± 0.000	0.002 ± 0.003	0.002 ± 0.003	0.001 ± 0.002	0.000 ± 0.001
2	0.004 ± 0.005	0.001 ± 0.002	0 ± 0	0.007 ± 0.005	0.006 ± 0.006	0.000 ± 0.001	0.001 ± 0.002	0.005 ± 0.005	0.004 ± 0.006	0.001 ± 0.002
3	0.004 ± 0.005	0.001 ± 0.002	0.006 ± 0.006	0 ± 0	0.001 ± 0.002	0.017 ± 0.014	0.000 ± 0.000	0.003 ± 0.004	0.012 ± 0.009	0.003 ± 0.004
4	0.001 ± 0.002	0.004 ± 0.005	0.006 ± 0.007	0.000 ± 0.001	0 ± 0	0.002 ± 0.003	0.005 ± 0.005	0.005 ± 0.005	0.002 ± 0.003	0.010 ± 0.009
5	0.007 ± 0.006	0.001 ± 0.002	0.006 ± 0.005	0.013 ± 0.014	0.005 ± 0.005	0 ± 0	0.005 ± 0.007	0.001 ± 0.003	0.005 ± 0.006	0.005 ± 0.006
6	0.009 ± 0.007	0.002 ± 0.003	0.003 ± 0.004	0.000 ± 0.000	0.005 ± 0.004	0.004 ± 0.004	0 ± 0	0.000 ± 0.000	0.001 ± 0.003	0.000 ± 0.001
7	0.001 ± 0.002	0.002 ± 0.003	0.003 ± 0.004	0.001 ± 0.002	0.006 ± 0.009	0.000 ± 0.000	0.000 ± 0.000	0 ± 0	0.003 ± 0.003	0.008 ± 0.005
8	0.008 ± 0.006	0.003 ± 0.004	0.008 ± 0.008	0.017 ± 0.010	0.006 ± 0.005	0.010 ± 0.007	0.003 ± 0.005	0.004 ± 0.005	0 ± 0	0.003 ± 0.004
9	0.001 ± 0.002	0.001 ± 0.006	0.002 ± 0.003	0.001 ± 0.002	0.013 ± 0.011	0.001 ± 0.003	0.000 ± 0.001	0.011 ± 0.007	0.001 ± 0.003	0 ± 0

6. The hardest samples are:



It is not surprising that these samples are hard to predict, we can see from visual inspection that the hardest samples have characteristics that make them inherently difficult to classify. Many of the hardest samples have simple but abnormal strokes that deviate from the normal form but end up replicating or nearly replicating the normal form of a different digit. One example of this is the sample in the 2nd rank which is labeled as a 3 but is in a slightly different shape to the rest of the 3s and can easily be mistaken for a 0. The sample in rank 3 is labeled as a 4 and has a very similar shape to a 9 despite the disconnection near the top. Additional examples include 3/5/8 as they all contain similar features which implies that their distinction is done near the decision boundaries. The test error shows what might be an irreducible error inherent in this dataset as some images may be written ambiguously.

7. **a)**

We choose set S based on the median heuristic that is commonly used in kernel methods. We set $c \approx 1/\text{median}(||\mathbf{x}_i - \mathbf{x}_j||^2)$. This ensures that for a typical pair of points, the kernel value is $e^{-1} \approx 0.37$ which provides a good ability to distinguish between classes. Our resulting median squared distance is approximately 244 (based on the subsets of the dataset) so we should use a $c \approx 0.004$. We will then use values in varying neighbourhoods of our median heuristic. We select small c which will use a very wide gaussian and increase to a large c which will use a narrow gaussian (will likely overfit) but expect the optimal range to be within a small neighbourhood of our heuristic.

$S = \{0.001, 0.01, 0.05, 0.1, 0.5, 1.0, 5.0\}$

7b)

Train and test error (mean $\pm$ std) for different values of c

c	Train Error	Test Error
0.001	0.0560 ± 0.0143	0.0747 ± 0.0155
0.010	0.0005 ± 0.0003	0.0280 ± 0.0034
0.050	0.0002 ± 0.0002	0.0394 ± 0.0052
0.100	0.0001 ± 0.0002	0.0553 ± 0.0051
0.500	0.0000 ± 0.0000	0.0687 ± 0.0071
1.000	0.0000 ± 0.0000	0.0699 ± 0.0059
5.000	0.0000 ± 0.0000	0.0709 ± 0.0058

7c)

Results for Q4 procedure over all c ∈ S

- Mean c*: 0.0100 ± 0.0000
- Train Error: 0.0005 ± 0.0003
- Test Error: 0.0282 ± 0.0040

7d)

Both kernels result in similar performance levels. This shows that neither has a significant advantage for the task. We can see that polynomial kernel is quite robust across d=4-7 whereas the gaussain kernel has significant sensitivity to c, the performance reduces quickly for c > 0.01. An interesting piece across both kernels is the fact that the sweet spot for hyperparameter tuning from cross validation matches what we found in the initial 20 runs. Our intuitions were right for both kernels, low polynomial degree and low c value do not allow the models to learn the data properly leading to high training errors whereas high polynomial degree and high c value allow the model to overfit and memorize the training data leading to bad generalization. We can interpret these parameters intuitively as higher d allows more feature interactions in the higher dimensional space and a higher c value creates a narrower gaussian allowing each point to be influenced only be very close neighbors. Cross validation yielded a very concrete answer for the gaussian kernel compared to the polynomial kernel as the gaussian kernel used c = 0.01 for all runs.

8. **a)**

One versus one trains $\binom{k}{2} = \frac{k(k-1)}{2}$ binary classifiers (one for each pair of classes). For each of the classifiers we only use training samples from the two classes that the classifier is assigned. During the training process we train the classifiers to distinguish between the pair of classes in its training data. This leads to training many classifiers but they each only operate on a pair of classes. The prediction phase is quite different from prior, it adheres to a voting procedure. For a new instance, run all the trained classifiers. They will each output a certain class (one of two) which we call a vote for that class. The final prediction is the class with the most votes (ties can be handled by random assignment or by confidence levels in the predictions). The advantage to this method is that each classifier is assigned a simpler problem than each OVR classifier, in OVO the classifiers only need to focus on features relevant to its local problem. The primary issue with this method is that it requires many classifiers, if we have 10 classes then we need 45 classifiers (10 choose 2). This increases at $O(k^2)$ which can be problematic for a dataset with 1000+ classes.

8b)

Results for OVO (one-versus-one) method:

Degree	Train Error	Test Error
1	0.0424 ± 0.0091	0.0703 ± 0.0104
2	0.0077 ± 0.0028	0.0413 ± 0.0040
3	0.0037 ± 0.0034	0.0362 ± 0.0048
4	0.0016 ± 0.0026	0.0340 ± 0.0063
5	0.0011 ± 0.0012	0.0338 ± 0.0037
6	0.0007 ± 0.0006	0.0332 ± 0.0046
7	0.0011 ± 0.0023	0.0364 ± 0.0052

8c)

Results for OVO cross validation:

- Mean d*: 4.85 ± 1.01
- Mean Train Error: 0.0009 ± 0.0008
- Mean Test Error: 0.0315 ± 0.0034

8d)

Both methods achieve solid performance but OVR outperforms OVO on average across the runs. We can see that OVR achieves around a 2.82% test score on its best parameter settings whereas OVO is only able to get to around 3.2% which is a relevant difference across numerous runs. Both methods handle the multi class problem quite differently and despite OVR's imbalance it generalizes better than OVO's balanced 1:1 approach. We also observe that the OVO selects a lower dimension on average from cross validation (mean of 4.85 vs 5.15), intuitively this makes sense as each classifiers job is to distinguish between two clases rather than one class from all of the others leading to lower complexity required. Interestingly, this increased complexity does not lead to improved performance on the test sets. One potential reason behind this is using hard voting instead of confidence scores in the classification, each classifier contributes an equal vote no matter the confidence in their vote. In terms of computational costs, OVO requires 45 classifier evaluations at prediction time compared to OVR's 10 which makes OVR far faster. Due to its accuracy and prediction complexity OVR seems to be a better choice for this dataset.

Part II

In this report, we discuss the performance of the following semi-supervised learning algorithms, Laplacian Interpolation and Laplacian Kernel Interpolation. The mean generalisation error and standard deviation are computed for four datasets of varying size ($m \in \{50, 100, 200, 400\}$), and with five levels of supervision, i.e. the number of labelled data points per class ($n \in \{1, 2, 4, 8, 16\}$.

Figures 1 and 2 present the mean estimated generalisation error and standard deviation for label propagation by Laplacian Interpolation and Laplacian Kernel Interpolation respectively.

```
=== Laplacian Interpolation (LI) ===
> # of known labels (per class)              1              2              4              8             16
v # of data points per label
50                               0.208 ± 0.132  0.165 ± 0.103  0.064 ± 0.039  0.052 ± 0.032  0.040 ± 0.019
100                              0.103 ± 0.143  0.045 ± 0.014  0.042 ± 0.009  0.033 ± 0.010  0.027 ± 0.011
200                              0.086 ± 0.100  0.029 ± 0.029  0.034 ± 0.030  0.018 ± 0.010  0.019 ± 0.005
400                              0.054 ± 0.075  0.029 ± 0.045  0.016 ± 0.014  0.012 ± 0.002  0.010 ± 0.002
```

Figure 1: Laplacian Interpolation Results

```
=== Laplacian Kernel Interpolation (LKI) ===
> known labels (per class)                 1              2              4              8             16
v # of data points per label
50                             0.129 ± 0.122  0.091 ± 0.062  0.045 ± 0.010  0.045 ± 0.017  0.037 ± 0.013
100                            0.094 ± 0.175  0.044 ± 0.009  0.039 ± 0.007  0.032 ± 0.010  0.028 ± 0.011
200                            0.030 ± 0.028  0.021 ± 0.010  0.021 ± 0.009  0.016 ± 0.004  0.018 ± 0.004
400                            0.017 ± 0.008  0.013 ± 0.005  0.014 ± 0.010  0.012 ± 0.002  0.010 ± 0.002
```

Figure 2: Laplacian Kernel Interpolation Results

Several observations can be made. Firstly, as the number of labelled data points per class n increases (and therefore the level of supervision), we see a consistent improvement in generalisation error both in mean and standard deviation for both methods. At $m = 50$, the mean error of Laplacian Interpolation drops from 0.208 at $n = 1$ to 0.040 at $n = 16$, and the respective standard deviations from 0.132 to 0.019. Likewise for Laplacian Kernel Interpolation, we see a drop in mean error from 0.129 at $n = 1$ to 0.037 at $n = 16$, and the respective standard deviation from 0.122 to 0.013.

Secondly, as the total per label dataset size m increases, we again see a consistent improvement in generalisation error both in mean and standard deviation for both methods. At $n = 1$, the mean error of Laplacian Interpolation drops from 0.208 at $m = 50$ to 0.054 at $m = 400$, and the respective standard deviations from 0.132 to 0.075. Likewise for Laplacian Kernel Interpolation, we see a drop in mean error from 0.129 at $m = 50$ to 0.017 at $m = 400$, and the respective standard deviation from 0.122 to 0.008.

Both methods appear to converge as $m, n \to \infty$, but in the sparse data regime we can clearly see that Laplacian Kernel Interpolation is superior, especially in mean error. At $n = 1, m = 50$ (the most sparse tested case), we observe a mean error for Laplacian Interpolation of 0.208 ± 0.132, compared to 0.129 ± 0.122 for Laplacian Kernel Interpolation.

The source of the high error in the sparse data regime is suspected to be limited graph connectivity - the graph is constructed using the k-nearest neighbour algorithm at $k = 3$, and at small m this could result in a disconnected graph, with islands of nodes. Furthermore with small n, it is possible that some of these islands are completely unlabeled.

In Laplacian Interpolation, we solve the following,

$$f_u = (D_{uu} - W_{uu})^{-1} W_{ul} f_l$$

Where f_u is the vector of the unlabeled nodes, f_l is the vector of the labeled nodes, D_{uu} is the diagonal submatrix for unlabeled node interactions, W_{uu} is the adjacency submatrix for unlabeled node interactions, and W_{ul} is the adjacency submatrix for unlabeled-labeled node interactions. The equation is taken from Zhu et al.[1]

In cases when we have a disconnected graph with fully unlabeled islands, the Laplacian becomes singular and so $(D_{uu} - W_{uu})^{-1}$ does not exist. In our implementation, we use the np.linalg.solve function to solve $(D_{uu} - W_{uu}) f_u = W_{ul} f_l$ which is equivalent, but when the Laplacian becomes singular this results in numerical explosions, causing the high errors in the sparse regime.

In Laplacian Kernel Interpolation, we solve the following,

$$\begin{aligned}
\boldsymbol{K} &:= (L^+_{ij} : i, j \in \mathcal{L}) \\
\boldsymbol{y}_{\mathcal{L}} &:= (y_i : i \in \mathcal{L}) \\
\boldsymbol{\alpha}^* &:= \boldsymbol{K}^+ \boldsymbol{y}_{\mathcal{L}} \\
\mathbf{v} &:= \sum_{i \in \mathcal{L}} \alpha_i^* \mathbf{e}_i^\top L^+
\end{aligned}$$

Where $\mathbf{v}$ is the prediction vector, L is the Laplacian matrix, $\mathcal{L}$ is the labeled node indices, y is label vector, and $+$ is the pseudoinverse operator. WIth this method, in the case of a singular Laplacian, the pseudoinverse provides an implicit regularisation. It uses Singular Value Decomposition to break down the Laplacian into invertible and non-invertible components and ignores the non-invertible ones. This results in lower errors in the sparse regime, when compared to Laplacian Interpolation.

Although Laplacian Kernel Interpolation performs better on these datasets, we might expect Laplacian Interpolation to be superior on large scale, well connected datasets. As we see in our experiment, as $m, n \to \infty$, the performance of the two algorithms converge. However, the benefit of Laplacian Interpolation at scale is in its computational efficiency.

Laplacian Kernel Interpolation requires computing the full pseudoinverse of the Laplacian, which is an $O(m^3)$ operation in time. Furthermore, storing the pseudoinverse L^+ itself requires $O(m^2)$ space in memory. For massive datasets, computing and storing the pseudoinverse can be prohibitively expensive.

Laplacian Interpolation on the other hand can be implemented by solving a sparse linear system of equations. There are solvers that exist that can do this efficiently, for example the Conjugate Gradient algorithm can be shown to be $O(m\sqrt{\kappa})$ in time, and $O(m)$ in space, where m is the number of non-zero entries and κ is the condition number (Shewchuk 1994[2]). This makes Laplacian Interpolation a much more scalable choice of algorithm.

[1] Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions, found at https://www.aaai.org/Papers/ICML/2003/ICML03-118.pdf

[2] An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, found at https://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf

Part III

1. (a) The sample complexity for the four algorithms (perceptron, winnow, least squares, 1-nearest neighbour) are plotted together in Figure 3.

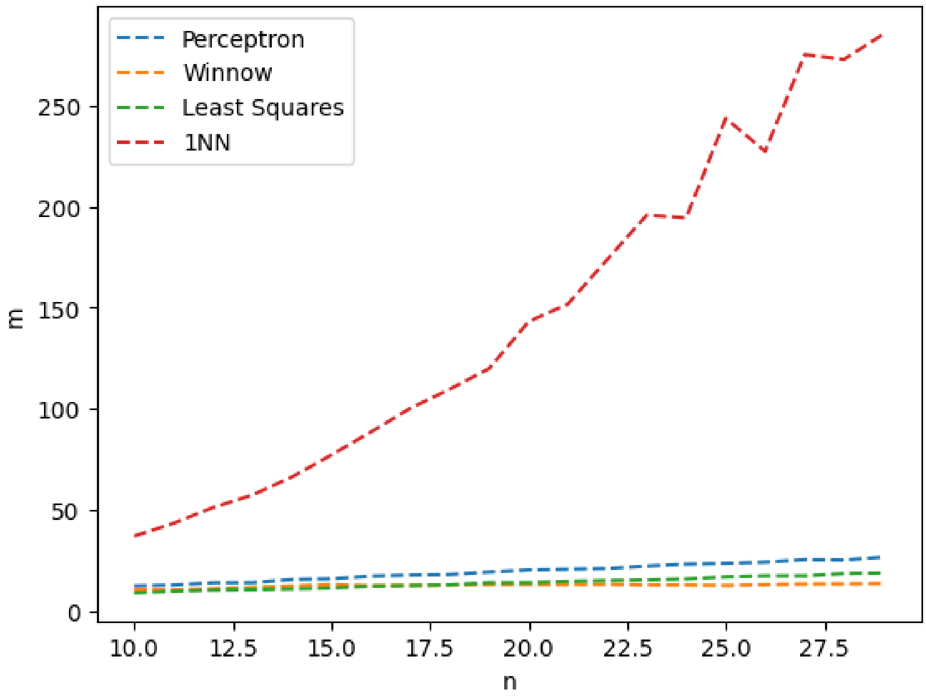


Figure 3: Sample Complexity plot [m against n]

For clarity, a plot with $1NN$ removed is provided in Figure 4, a log-scale plot is plotted in Figure 5 and a log-scale plot with $1NN$ removed is plotted in Figure 6.

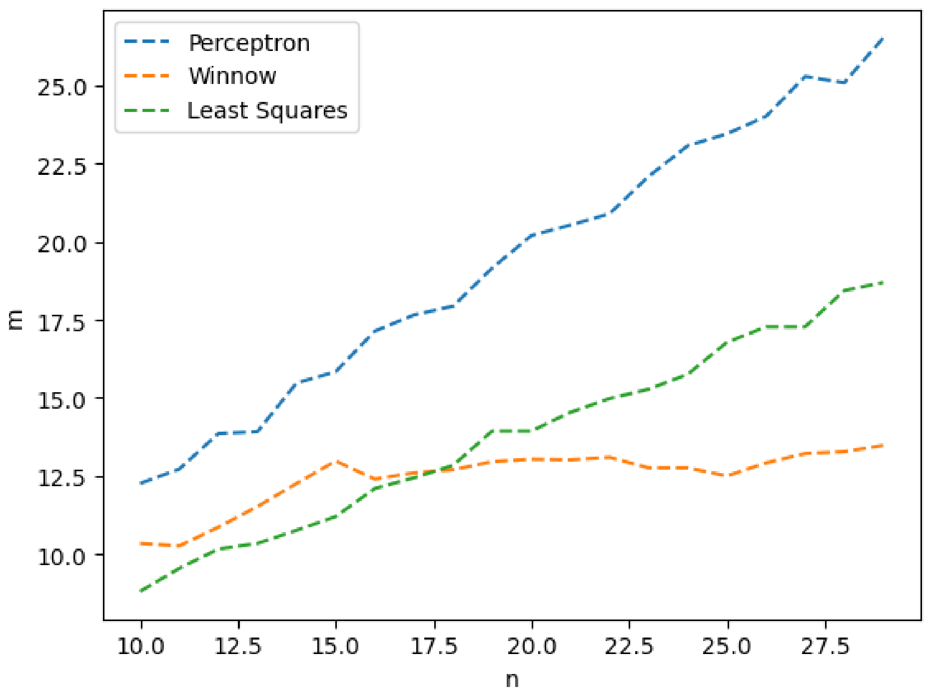


Figure 4: Sample Complexity plot [m against n]

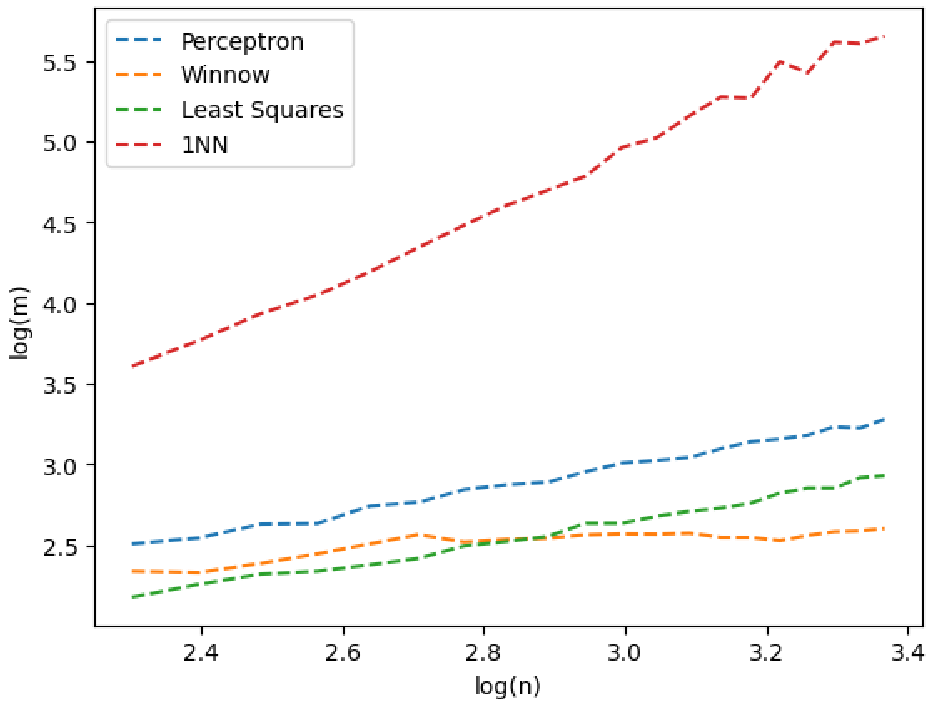


Figure 5: Sample Complexity plot [log(m) against log(n)]

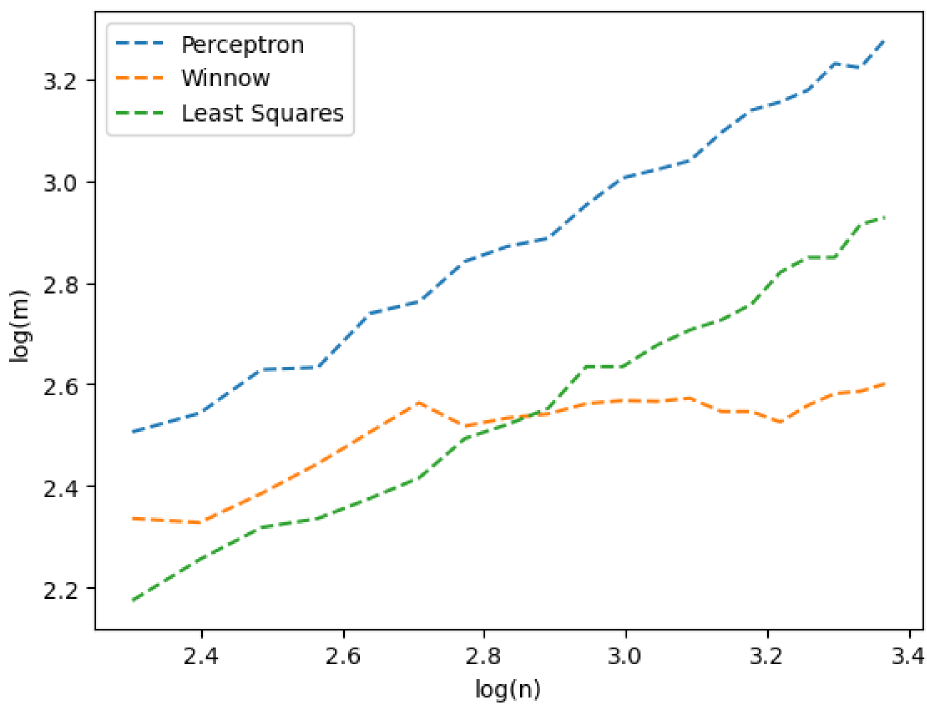


Figure 6: Sample Complexity plot [log(m) against log(n)]

(b) Calculating the sample complexity exactly can be done by the following:

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) := 2^{-n} \sum_{\boldsymbol{x} \in \{-1,1\}^n} I[\mathcal{A}_{\mathcal{S}}(\boldsymbol{x}) \neq x_1]$$

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

$$\mathcal{C}(\mathcal{A}) = \min\left\{ m \in \{1,2,\dots\} : \Big(2^{-nm} \sum_{S \subseteq \{-1,1\}^{nm}} \mathcal{E}(\mathcal{A}_{S})\Big) \leq 0.1 \right\}$$

Where $\mathcal{C}(\mathcal{A})$ is the sample complexity, and $\mathcal{E}(\mathcal{A}_{\mathcal{S}})$ is the generalisation error. $\mathcal{A}_{\mathcal{S}}(x)$ is the prediction of algorithm $\mathcal{A}$ trained from data sequence $\mathcal{S}$ on pattern x.

This is computationally infeasible as the number of combinations per data point m grows at a rate of 2^n. Instead of computing the expectation $\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})]$ exactly, we first make an estimate of $\mathcal{E}(\mathcal{A}_{\mathcal{S}})$ then recalculate over many Monte Carlo simulations to compute the expectation. $\mathcal{E}(\mathcal{A}_{\mathcal{S}})$ is estimated by the mean classification error over 50 hold out test samples, and this is repeated 100 times from which the mean is taken to give the expectation. Since the expectation of the generalisation error is expected to decrease monotonically with m, we iteratively compute the sample complexity with increasing m, starting from $m = 1$, until the threshold 10% error is reached. Finally, rather than increasing m in increments of 1, it is increased in increments of 2 for additional computational efficiency.

There are a few tradeoffs and biases in the method used to calculate the sample complexity. In the estimation of $\mathcal{E}(\mathcal{A}_{\mathcal{S}})$, we only use 50 test samples, which will result in some bias in the form of a sampling error. The larger the test set, the lower the sampling error, but the higher the computational cost. A fixed test set regardless of the size of the training set m (as opposed to a percentage of the training set), is used to ensure stable estimates even at low values of m. Another sampling error is induced in the Monte Carlo simulation used to estimate the expectation. The more repeats we conduct, the lower this sampling error but again at a higher computational cost. Using a step size for incrementing m induces another bias, in the form of a discretisation error - which is large especially when m is small (relative to the step size). The smaller the step size, the smaller this discretisation error.

(c) For the perceptron and least squares algorithms, Figures 4 and 6 clearly show near linear growth of m with n and $log(m)$ with $log(n)$ respectively. This growth is also roughly at a rate of 1, and so approximately, $m \propto n$. Therefore, for these algorithms, $m = \Theta(n)$.

For winnow, Figure 4 shows a flattening curve, and Figure 5 shows a curve with a clearly shallower slope than the perceptron and least squares curves. It is clearly sub-linear, and since the curve in Figure 6 seems to be flattening, a $m = \Theta(log(n))$ relationship is suggested.

For 1NN, Figure 3 shows that m explodes in size as n increases. Furthermore, Figure 5 shows a linear curve with a steeper slope than perceptron and least squares. This would suggest a polynomial relationship with n, $m \propto n^k$. However, it is difficult to distinguish between polynomial growth and exponential growth with this limited sample size of n, and so we can only give the empirical lower bound suggestion (as opposed to a tighter bound), $m = \Omega(n^k)$.

Therefore winnow is the most efficient algorithm and should be used for large n, and 1NN especially should be avoided.

(d) For some arbitrary sequence of examples,

$$S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathbf{R}^n \times \{-1, 1\}$$

The number of mistakes M is bounded by the Novikoff Perceptron bound as follows,

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

With $R := max_t||\boldsymbol{x_t}||$ when there exists a vector $\boldsymbol{v}$ with $||\boldsymbol{v}|| = 1$ and constant γ such that $(\boldsymbol{v} \cdot \boldsymbol{x_t})y_t \geq \gamma$.

In this problem, $\boldsymbol{x}$ is drawn from $\{-1, 1\}^n$, and so the squared norm is given by $||\boldsymbol{x}||^2 = \sum_{i=1}^n x_i^2 = n$. Therefore, $R = \sqrt{n}$. The target function here is $y = x_1$, which corresponds to a target weight vector $\boldsymbol{v}$ of $[1, 0, 0, ..., 0]$. Thus, $(\boldsymbol{v} \cdot \boldsymbol{x_t})y_t = x_{1,t}y_t = y_t^2 = 1$ as $y \in \{-1, 1\}^n$. Therefore, $\gamma \leq 1$.

Substituting in, and using $\gamma = 1$ to achieve the tightest bound, we have,

$$M \leq \left(\frac{\sqrt{n}}{1}\right)^2 = n$$

So regardless of the sequence length, m, the number of mistakes is bounded by n. Therefore, the bound for the probability of a mistake on the s_{th} example is given by the number of mistakes divided by the number of examples,

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

(e) The 1NN algorithm predicts by assigning the label of the closest training point in Euclidean distance on an n-dimensional boolean hypercube.

$$\hat{y} = \min_t D(\boldsymbol{x}^{test}, \boldsymbol{x}^{train,t}) = \min_t ||\boldsymbol{x}^{test}, \boldsymbol{x}^{train,t}||_2 = \min_t \sqrt{\sum_{j=1}^n (x_j^{test} - x_j^{train,t})^2}$$

$$\boldsymbol{x} \in \{-1, 1\}^n$$

This hypercube has 2^n vertices, and the prediction is only correct when the closest training point shares its first coordinate with the test point,

$$\hat{y} = y \text{ when } x_1^{test} = \min_t x_1^{train,t}$$

In the case of an incorrect prediction, the distance contribution of the first coordinate is 2. As $n \to \infty$ this contribution becomes negligible compared the to expected distance contribution of the remaining $n - 1$ dimensions. Therefore for fixed m, as $n \to \infty$ the generalisation error $\varepsilon(\mathcal{A}_\mathcal{S}) \to \infty$.

In order to ensure $\varepsilon(\mathcal{A}_{\mathcal{S}}) < 0.1$ even as n grows large, m must also grow to ensure that for any arbitrary test point, the nearest neighbour shares its first coordinate with the test point. For this to be true, the number of training points m should grow proportionally to the number of vertices of the n-dimensional hypercube, i.e. 2^n. Because the training samples are generated *iid*, a higher number of training points may be required. However, the above argument gives us a lower bound estimate $m = \Omega(2^n)$.