Q1 Solution

Part 1: One vs Rest approach (OvR)

The OvR (One versus Rest) approach divides and simplifies the problem of multi-class classification into being able to tell each class distinctly from all the other classes.

1. TRAINING: Formally, given K classes, OvR trains K models. For each model, the training data is updated such that the k-th class labels are kept as-is while all the other class labels are grouped into the "other" set. Let's assume the k-th class is the "positive" class while training, with the "all other" set being the "negative" class. The usual Binary Perceptron algorithm is then used to train the model.

2. PREDICTION: Given we have K models, a "maximisation" approach is used. We let the K models compute a "confidence" score (called as "activation") for the positive class on the test/evaluation data. Then, the model with the highest activation score is selected - and the corresponding label (k) is predicted for the test data.

Part 2: Kernel-based Perceptron OvR: Implementation Details

2.1 Kernel Perceptron Algorithm

1. **Representation**: A vector **alpha** of length N (number of data points) is initialised to 0 to track the weight coefficients of all data points

2. **Evaluation**

   (a) The Gram Matrix is pre-calculated for training data via matrix multiplication (XX^T), capturing the kernel evaluations for all combinations of data points (NxN)

   (b) For each data point x_i, w(.) is evaluated as the dot product between the alpha vector and the i-th row of the Gram matrix (the kernel vector for x_i)

3. **Prediction and Update**

(a) Prediction: For each data point x_i, label prediction is based on sign[w(x_i)] as: if $w(.) > 0$ then 1 else -1

(b) Update: If the prediction is incorrect, we update the corresponding co-efficient $alpha_i$ by adding the correct label y_t, that is: $alpha_i$ updates to $alpha_i + y_t$

(c) This update can be justified by expanding the provided update: $w_{t+1}(.) = w_t(.) + alpha_t * K(x_t, .)$ and showing it is equivalent to adding $alpha_t$ to the co-efficient of $K(x_t, .)$, that is: adding y_t (label) to the alpha vector element corresponding to the data point being currently 'seen'

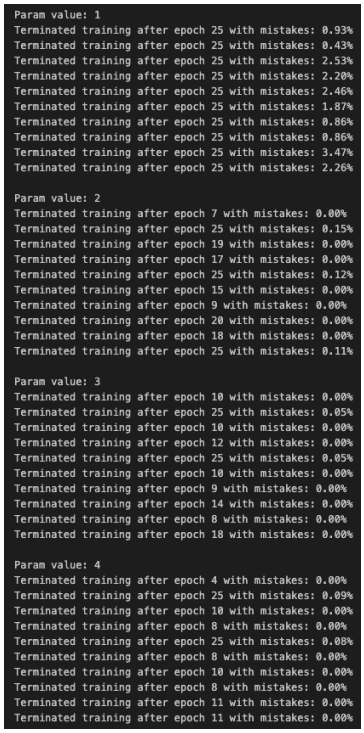
2.2 Observations on selecting max epoch for training

We set the maximum epoch limit as follows:

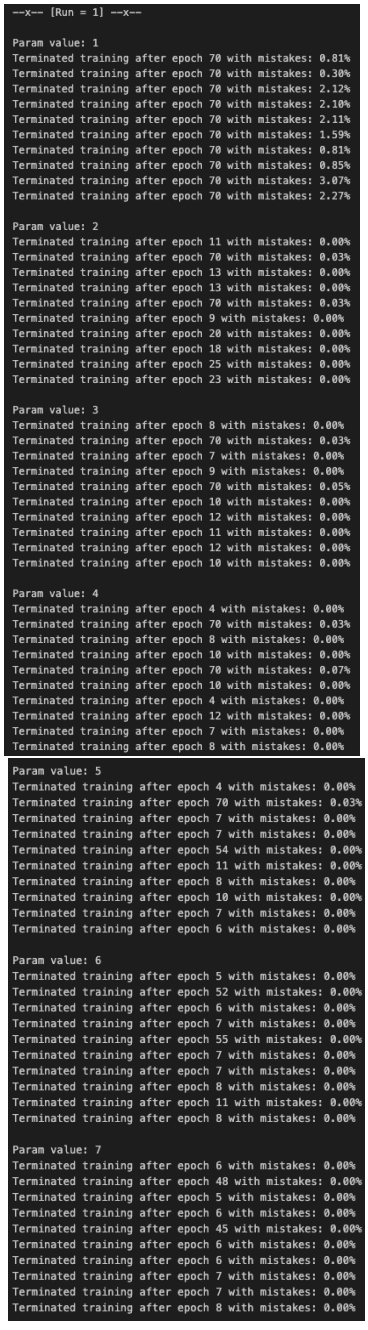
- Polynomial - OvR: 25
- Gaussian - OvR: 20 (Gaussian kernel is more powerful to Polynomial kernel, leading to earlier convergence on average, all other factors being similar)
- Polynomial - OvO: 20 (OvO trains one-vs-one models, and thus has simpler classification problems at the model level and thus earlier convergence)

Selecting the maximum epoch limit:

1. Number of epochs required for convergence decreases with increasing model capacity. Simpler models are just unable to resolve the remaining few errors/mistakes, leading to no or late convergence. Therefore, we are **bounded by the simplest model(s)** - for example, d = 1,2 for polynomial kernel.
2. Most converging runs did so by 25-30 runs, whereas the distribution of error rates did NOT change for the simplest model(s) across maximum epochs range of {25 - 70} (final error rate results shared below for a) Polynomial OvR and b) Gaussian OvR cases)



(a) polynomial ovr w/ max epoch = 25



(b) polynomial ovr w/ max epoch = 70

Figure 1: Gaussian OvR: 20 vs 30

```
--x-- [Run = 2] --x--

Param value: 0.005
Terminated training after epoch 12 with mistakes: 0.00%
Terminated training after epoch 20 with mistakes: 0.08%
Terminated training after epoch 9 with mistakes: 0.00%
Terminated training after epoch 7 with mistakes: 0.00%
Terminated training after epoch 20 with mistakes: 0.00%
Terminated training after epoch 9 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 13 with mistakes: 0.00%
Terminated training after epoch 20 with mistakes: 0.08%
Terminated training after epoch 20 with mistakes: 0.11%

Param value: 0.01
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 15 with mistakes: 0.00%
Terminated training after epoch 7 with mistakes: 0.00%
Terminated training after epoch 7 with mistakes: 0.00%
Terminated training after epoch 13 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 10 with mistakes: 0.00%
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 10 with mistakes: 0.00%

Param value: 0.05
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%

Param value: 0.1
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
```

```
Param value: 0.5
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%

Param value: 1.0
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%

Param value: 2.0
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
```

(a) gaussian ovr w/ max epoch = 20

```
--x-- [Run = 1] --x--

Param value: 0.005
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.05%
Terminated training after epoch 10 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.09%
Terminated training after epoch 23 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 16 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 12 with mistakes: 0.00%

Param value: 0.01
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.07%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.09%
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 8 with mistakes: 0.00%

Param value: 0.05
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 7 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 7 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%

Param value: 0.1
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
```

```
Param value: 0.5
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 6 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%

Param value: 1.0
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%

Param value: 2.0
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 4 with mistakes: 0.00%
Terminated training after epoch 3 with mistakes: 0.00%
Terminated training after epoch 5 with mistakes: 0.00%

--x-- [Run = 2] --x--

Param value: 0.005
Terminated training after epoch 8 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.11%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 15 with mistakes: 0.00%
Terminated training after epoch 30 with mistakes: 0.08%
Terminated training after epoch 14 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 11 with mistakes: 0.00%
Terminated training after epoch 14 with mistakes: 0.00%
Terminated training after epoch 10 with mistakes: 0.00%
```

(b) gaussian ovr w/ max epoch = 30

Figure 2: Gaussian OvR: 20 vs 30

Part 3: Polynomial Kernel OvR Perceptron (20 run avg., Train-Test)

3.1 Results

d	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Train Error Mean	5.4315	0.1025	0.0450	0.0595	0.0140	0.0145	0.0145
Train Error S.D.	1.2265	0.3017	0.1291	0.2100	0.0190	0.0147	0.0216
Test Error Mean	9.0625	3.1745	2.9640	2.8815	2.8905	2.9795	3.0925
Test Error S.D.	1.3443	0.3463	0.2918	0.3448	0.3414	0.3554	0.3352

Table 1: Results for Q3

3.2 Observations

1. Train error mean decreases with increasing d (complexity of feature space/ capacity of model)
2. Test error mean for d=1 is quite high (9-10 percent) compared to d=2 onwards
3. Test error mean decreases initially (from d = 1 to 4) and then starts increasing again (from d = 4 to 7). Model begins to overfit on the data as capacity increases beyond d=4

Part 4: Polynomial Kernel OvR Perceptron (CV)

4.1 Results

Mean best param (d)	4.60
SD best param (d)	1.05
Mean Train Error	0.0105
SD Train Error	0.0094
Mean Test Error	2.7505
SD Test Error	0.3859

Table 2: Results for Q4

4.2 Observations

1. Best hyper-parameter value for polynomial kernel is "around 4" (3-5)
2. Test error mean is 2.75 percent, with .40 percent std. dev. (approx, 2.5-3 percent average error rate)
3. This CV-based selection helps prevent overfitting, which is seen in "bigger" models (higher d)

Part 5: Confusion Matrix

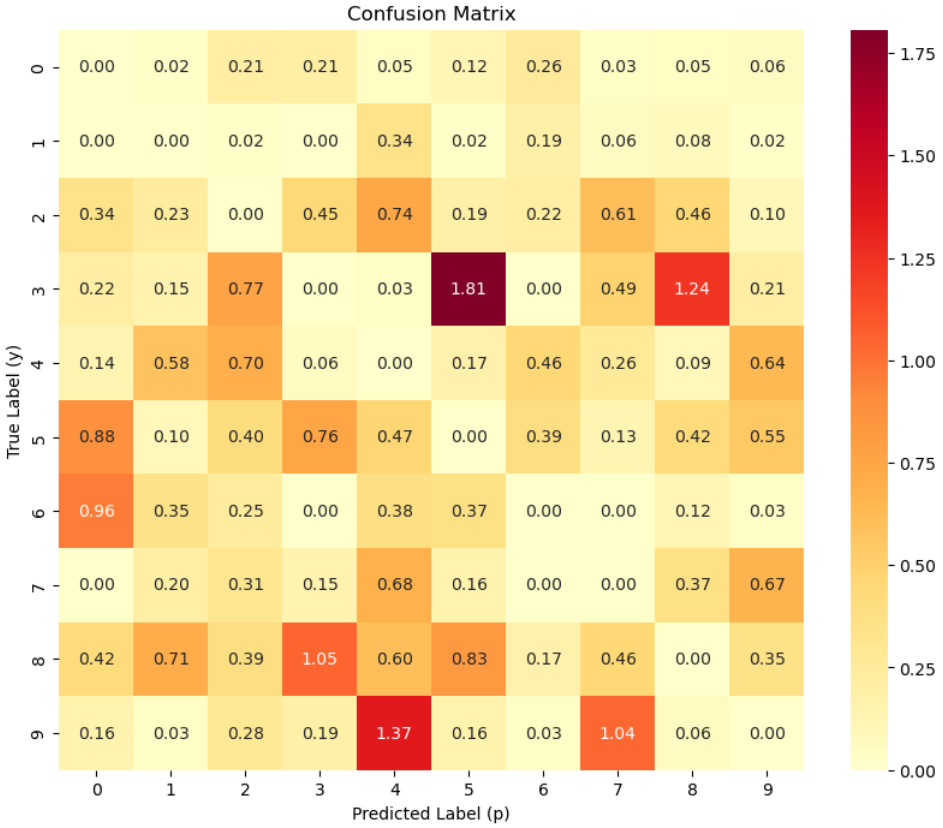


Figure 3: polynomial ovr: confusion matrix

5.1 Observations

1. **Misclassified Digits**
   (a) Digits 0 and 1 are rarely misclassified
   (b) 8s are misclassified as other digits approx. uniformly/equally, and is the most misclassified digit
2. **Most Common Misclassification(s)**

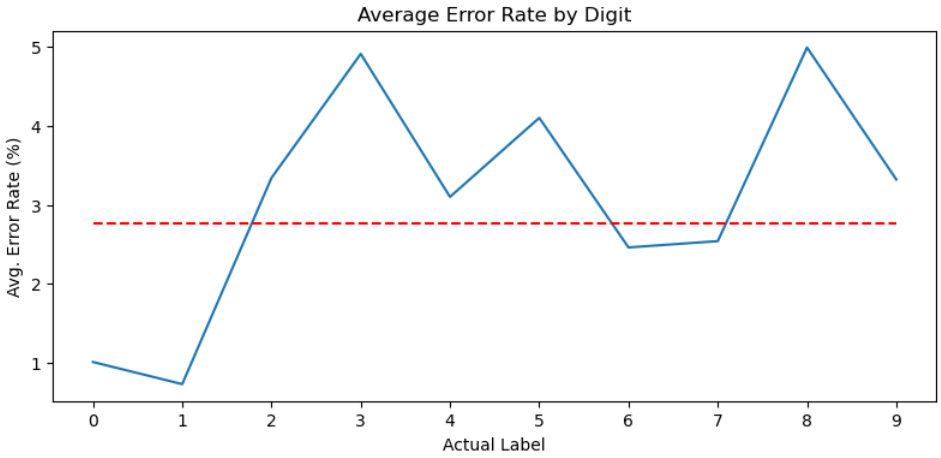


Figure 4: polynomial ovr: avg. error rate by digit [red line: avg. across digits]

(a) {3,5,8} - most common set

- (3,5): 1.8 percent of 3s are misclassified as 5
- (8,3): 1.25 percent of 8s are misclassified as 3
- (3,8): 1 percent of 3s are misclassified as 8

(b) {4,9} - second-most common set

- (9,4): this misclassification occurs with 1.4 percent error rate

Part 6: Hardest to Predict Samples

6.1 Observations on hardest samples

1. Hardest 5 samples were identified as those with smallest margin (activation difference) in OvR prediction b/w models for the top 2 most-probable digits predicted

   (a) Some digits are drawn similar to other digits (2: 0, 8 or 6: 4)

(b) Some of the digits are incompletely drawn (8, 2)

2. Observations

   (a) It is not surprising to see most of the confusions based on the structure of our models. Some digits are structurally similar to others and slight variations in their shape can lead to confusion (e.g. 3 v 8, 1 v 7, etc.) for the perceptron models, although it is still 'surprising' to see the confusion as a human looking at some of the images. For example, across the models, it is naturally harder for a model to differentiate b/w 3 vs 8, compared to 3 vs 4

   (b) The 3rd sample is a bit more surprising. The 6 is clearly legible for a human. However, given the shape of the loop being same as that of an 8 and the upper stroke of the 6 being almost "closed" like an 8, we can see how the test data point might lie somewhere close to the decision boundary of the perceptron models

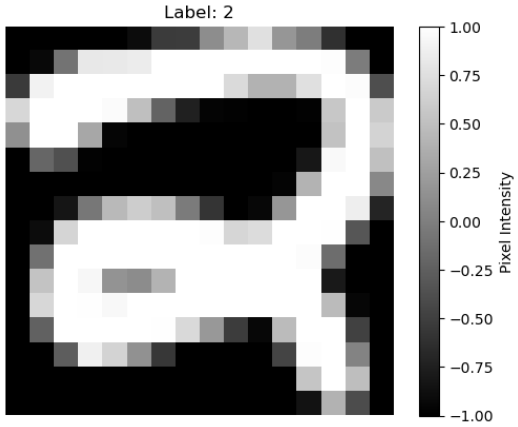


Figure 5: hardest 1

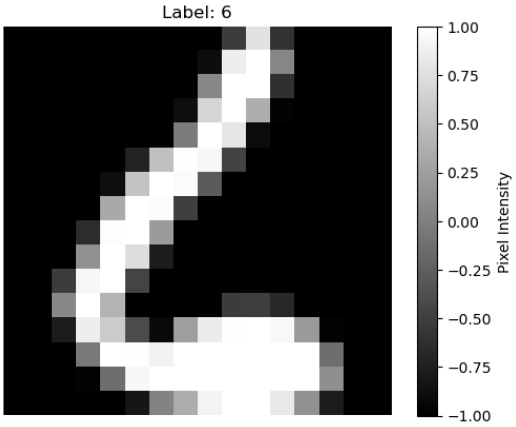


Figure 6: hardest 2

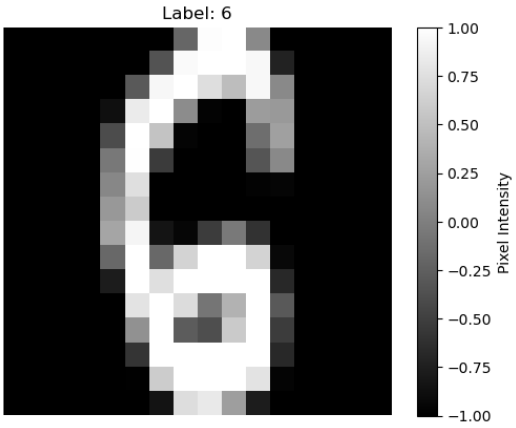


Figure 7: hardest 3

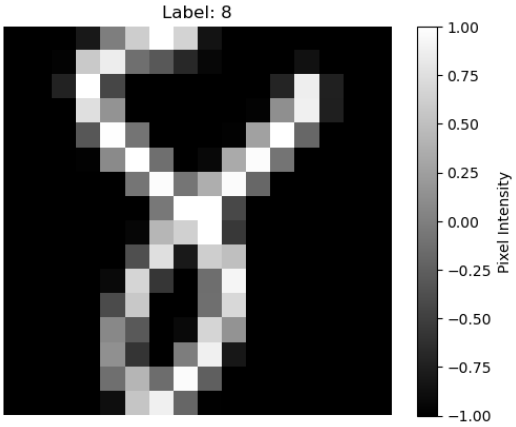


Figure 8: hardest 4

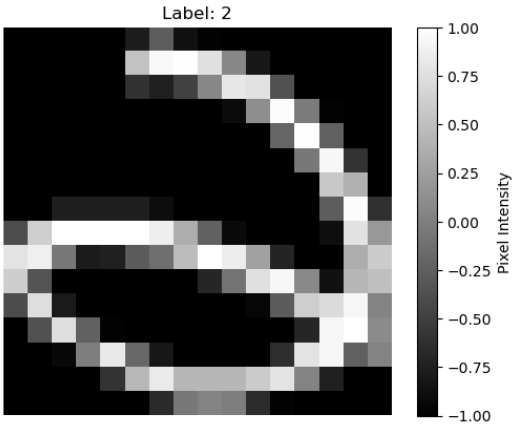


Figure 9: hardest 5

Part 7: Gaussian Kernel OvR Perceptron

7.1 7a: Parameter Set for Gaussian Kernel

1. Gaussian Kernel: $exp\{-c * ||p - q||^2\}$. Exponent value for $1/e(...)$ increases with increasing c. Two vectors specific distance apart lead to larger exponent. Therefore, Gaussian Kernel function value decreases with increasing c for a {p,q}. As c increases, even close vectors seem more apart.

2. Too low values of c will lead to underfitting as differences in pixel data shirnks, leading to "loss" of data. Too high values of c would lead to small differences (natural variations) in pixel data to be seen by the model as very different things

3. An initial test was conducted with $c = 1.0$. This resulted in significantly higher test data errors compared to training data errors, indicating a strong tendency toward overfitting at this scale. Therefore, the range was selected to prioritise smaller values of c

4. Parameter set S: [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1.0, 2.0]

7.2 7b: Gaussian OvR: Train-Test Results

c	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**	**0.5**	**1.0**	**2.0**
Train Error Mean	1.9890	0.2240	0.0275	0.0000	0.0000	0.0000	0.0000	0.0000
Train Error SD	0.9793	0.8963	0.0681	0.0000	0.0000	0.0000	0.0000	0.0000
Test Error Mean	5.4205	3.3020	2.7145	3.8565	5.3990	6.8790	10.7900	34.7960
Test Error SD	1.1011	1.1676	0.4580	0.4052	0.4203	0.5989	0.8314	0.7130

Table 3: Results for Q7b

7.3 7c: Gaussian OvR: CV Results

Mean best param (c)	0.01
SD best param (c)	0.00
Mean Train Error	0.0645
SD Train Error	0.2323
Mean Test Error	2.6565
SD Test Error	0.7041

Table 4: Results for Q7c

7.4 7d: Polynomial vs Gaussian OvR

1. Gaussian kernel has slightly lower error rate on average for best parameter (0.01 gaussian vs 4 polynomial)

2. Gaussian kernel has lower error rate on average for simpler models (0.1 percent lower) BUT higher error rate for overly-complex models. Gaussian kernel perceptron is more prone to overfitting (conditioned on the range of hyperparameter values selected for Gaussian kernel)

Part 8: Polynomial Kernel OvO Perceptron

8.1 8a: Research and Explain OvO approach

The OvO (One versus One) approach divides and simplifies the problem of multi-class classification into being able to tell each class distinctly from each other class, separately - that is, pairwise classification models

1. TRAINING: Formally, given K classes, OvO trains K*(K-1)/2 models. For each model (for each class-pair), the training data is filtered for the two classes in the current pair. The usual Binary Perceptron algorithm is then used to train each model.

2. PREDICTION: Each of the K*(K-1)/2 outputs a class prediction for the test data. The predictions are taken and tallied as votes for the K classes. The class with the highest overall number of votes is selected as the final prediction for the test data

8.2 8b: Polynomial OvO: Train-Test Results

d	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Train Error Mean	1.3900	0.0785	0.0490	0.0405	0.0190	0.0195	0.0200
Train Error SD	0.4979	0.1330	0.1188	0.0558	0.0194	0.0167	0.0348
Test Error Mean	5.7155	3.3475	3.1580	3.1620	3.1210	3.3365	3.5425
Test Error SD	0.7043	0.5021	0.4819	0.4802	0.4855	0.5760	0.5293

Table 5: Results for Q8b

8.3 8c: Polynomial OvO: CV Results

Mean best param c	4.10
SD best param	1.17
Mean Train Error	0.0345
SD Train Error	0.0431
Mean Test Error	3.1385
SD Test Error	0.4368

Table 6: Results for Q8c

8.4 8d: Polynomial OvR vs OvO

1. On average, slightly higher error rate (except d=1, simplest model) to polynomial OvR approach

2. OvR models see full data at once v/s OvO models that see data points for just two classes at a time could explain why generalisation is slightly better.

3. OvO has higher error rate on average for best parameter (for 4-degree polynomial kernel function)

4. Even though the OvO approach learns an order of K (number of classes) more models, as the data is significantly less per model (2*N vs K*N, assuming

uniform distribution) AND the confusions are less per model (6 vs 3 is simpler to classify than 6 vs 3,5,8, for example). OvO trains faster on similar data with the same perceptron kernel and hyperparameter value. [5m 45s vs 16m 41s, almost 3 times faster]

Q2 Solution

Part 1

Table 7: Errors and standard deviations for semi-supervised learning via Laplacian interpolation (upto 3 decimal places)

# data points per label	# of known labels (per class)				
	1	2	4	8	16
50	0.265 ± 0.123	0.147 ± 0.103	0.083 ± 0.068	0.045 ± 0.012	0.041 ± 0.013
100	0.052 ± 0.030	0.038 ± 0.010	0.046 ± 0.020	0.035 ± 0.010	0.029 ± 0.012
200	0.068 ± 0.091	0.061 ± 0.084	0.020 ± 0.006	0.020 ± 0.005	0.016 ± 0.006
400	0.107 ± 0.146	0.023 ± 0.020	0.014 ± 0.005	0.012 ± 0.003	0.009 ± 0.002

Table 8: Errors and standard deviations for semi-supervised learning via Laplacian (Kernel) interpolation (upto 3 decimal places)

# data points per label	# of known labels (per class)				
	1	2	4	8	16
50	0.092 ± 0.060	0.069 ± 0.040	0.052 ± 0.024	0.042 ± 0.008	0.038 ± 0.014
100	0.042 ± 0.007	0.036 ± 0.009	0.042 ± 0.008	0.033 ± 0.010	0.026 ± 0.014
200	0.021 ± 0.007	0.020 ± 0.005	0.018 ± 0.004	0.018 ± 0.005	0.016 ± 0.006
400	0.027 ± 0.053	0.017 ± 0.014	0.012 ± 0.002	0.011 ± 0.001	0.009 ± 0.002

Part 2

Observations about Table 7 (LI):

- For each dataset size (50, 100, 200, 400 per class), the error decreases as the number of labels per class l increases, then flattens once it is moderately large.
- With more unlabeled data (200 and 400 per class), the methods reach very low error (around 1-2%) with enough labels, showing that the graph captures the manifold structure and label smoothness well

- Standard deviations across the 20 runs shrink as l grows, which is expected because more labels per class reduce the randomness introduced by sampling the labeled set
- There is a minor increase in error going from 2 to 4 known labels in the same dataset (200 data points per label), but it is within the standard deviation and is not statistically significant.

Additional observations about Table 8 (LKI):

- Overall error levels are very similar to LI, typically in the low single-digit percent range once $l >= 2$
- For large graphs and very few labels (e.g. 200/400 per class with l=1), LKI achieves noticeably lower mean error than LI, and with smaller variance.
- As l increases, LKI and LI curves nearly coincide, indicating that with enough labeled points both converge to very similar solutions.

Part 3

Comparison of LI and LKI:

- The accuracy for both LI and LKI is similar, with LKI generally having slightly lower error rates.
- Specifically, when the number of known labels per class is low (1-2) LKI significantly outperforms LI.
- When there are very few labels, the direct harmonic solution can be more sensitive to where those labels lie on the graph, whereas the kernel interpolation implicitly regularises the norm of the interpolant, which can lead to better generalisation and lower variance with scarce labels.
- As the number of labelled points increases, both methods converge to similar solutions.

Part 4

For the given dataset, LKI tends to slightly outperform LI, with LI being the "weaker" method. The given dataset is simple, and the labeled examples are labeled correctly. In the case where some labels are labeled incorrectly, LKI might propagate those errors further than LI and this might lead to worse performance.

Also, graphs where the Kernel matrix is nearly singular, for example if the number of labeled points is very high compared to the size of the graph and the labeled points are concentrated together, α^* might be numerically unstable and simple LI might be preferred over LKI.

Q3 Solution

(a) Sample Complexity against Dimension for Different Algorithms

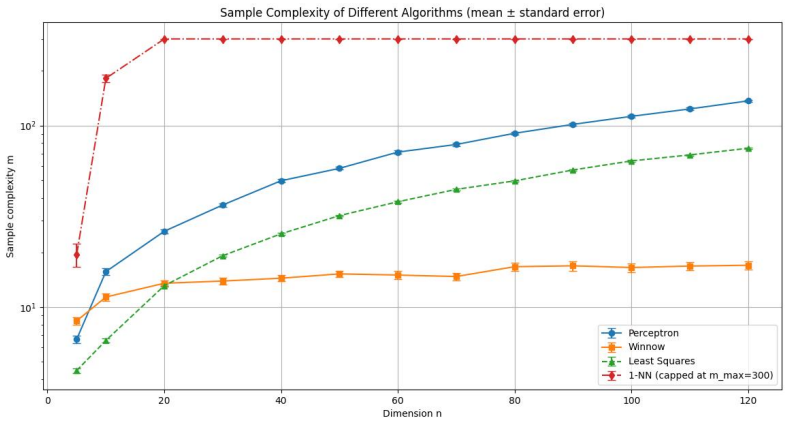


Figure 10: Sample complexity as a function of the dimension n for different algorithms.

As shown in Figure 10, the sample complexity generally increases with the dimension of X, and different algorithms exhibit different growth rates.

To ensure the reliability of these results:

- The data points represent the mean sample complexity across multiple independent trials, with error bars indicating the **Standard Error (SE)**. This provides a measure of how well the sample mean estimates the true population mean.
- For the 1-NN algorithm, a cap of $m_{max} = 300$ was imposed. Because the error rarely dropped below 10% within this range for high n, the red curve effectively represents a **lower bound** on the true sample complexity, illustrating the algorithm's total failure in high-dimensional sparse tasks.
- The semi-log scale makes it clear that while Perceptron and Least Squares grow linearly, Winnow's complexity is nearly invariant to n, appearing almost constant on this scale.

(b) Estimating Sample Complexity

(i) Method

To estimate the sample complexity, we used a Monte Carlo simulation approach:

- Generated a training data matrix X_{train} of size $m_{\max} \times n$, with $m_{\max}$ samples and dimension n, where each entry is sampled uniformly from $\{-1, 1\}$. Labels were defined as $y_i = x_{i,1}$.
- Evaluated algorithms at dimensions increasing in increments of 10 (i.e., $n = 10, 20, 30, \ldots$) to reduce computational cost.
- For a fixed dimension n, started with a single training sample ($m = 1$), trained the algorithm, and estimated the generalisation error.
- Incremented the sample size by 1, repeating training and evaluation until the generalisation error fell below 10%. If the generalization error does not fall below 10%, we stop at m_{max} which in this case is 300
- Estimated the generalisation error using a separate test set of 7,500 samples generated from the same distribution.
- Repeated this procedure n_{trials} times, averaging the resulting m values to obtain the expected sample complexity. To save computational cost, we used the same test set for estimating the generalization error across different trials. For the 1-NN algorithm, this was repeated 20 times; for other algorithms, 150 times for small n and 75 times for large n.
- Vectorized operations in the Winnow and 1-NN implementation to save computation time

Note that we did not use Hoeffding's inequality here, since it provides a high-probability bound on the generalisation error of a single fixed hypothesis. In our setting, for each dimension we train multiple hypotheses on independently sampled training sets and estimate the expected generalisation error via Monte Carlo averaging. As a result, empirical averaging is more appropriate than applying a concentration bound to individual hypotheses.

(ii) Trade-offs and Biases

This estimation method involves trade-offs between accuracy and computation time:

- Using a finite test set introduces noise in the generalisation error estimate. Larger test sets reduce noise but increase computation. Reusing the same test across trials saves computation time and lowers the variance of the error estimate for each trained model, at the cost of introducing correlation across trials.
- Incrementing the sample size by 1 may slightly overestimate the true minimum number of samples required for 10% error.
- Evaluating dimensions in steps of 10 may miss finer changes in sample complexity.
- Averaging over multiple Monte Carlo trials reduces variance, but more trials increase computational cost.
- For large n, training and evaluating algorithms—especially 1-NN—becomes expensive, limiting the feasible range of dimensions and sample sizes.
- Defining sample complexity as the first m where error falls below 10% makes the estimate sensitive to small fluctuations around this threshold.

(c) Estimating Sample Complexity against Dimension

Based on the experimental data shown in Figure 10 and the analytical characteristics of the 1-sparse target concept (where the label $y = x_1$), we estimate the growth rates of the sample complexity m as follows:

- **Winnow:** $m = O(\log n)$
  For a 1-sparse target concept ($k = 1$), Winnow's theoretical mistake bound is $M \leq 2 \ln n$. Since the sample complexity m required to reach an error ϵ is proportional to M/ϵ, our experimental results confirm this logarithmic scaling. Winnow effectively ignores irrelevant dimensions via its multiplicative update rule.

- **Least Squares:** $m = \Theta(n)$. In the underdetermined case ($m < n$), we define the classifier via the weight vector

$$\mathbf{w} = X^{+}\mathbf{y},$$

where X^{+} is the pseudoinverse. This finds the minimum-norm solution consistent with the training data. However, since the labels $\mathbf{y}$ are determined by only one coordinate (x_1) while X contains $n-1$ noise dimensions, the pseudoinverse solution effectively distributes the "importance" across all dimensions to minimize the norm. For the signal w_1 to dominate the noise components and achieve a 10% generalization error, the number of samples m must scale linearly with the total dimension n to sufficiently constrain the system. This linear growth is clearly visible in the experimental results (Figure 10).

- **Perceptron:** $m = O(n)$
Based on **Perceptron Mistake Bound**, the mistake bound is $(R/\gamma)^2$. For this problem, $R^2 = n$ and $\gamma = 1$, yielding a bound of n. The linear trend in Figure 1 aligns with the online-to-batch conversion principle, where generalization error $\epsilon \approx E[M]/m$.

- **1-Nearest Neighbor (1-NN):** $m = \Omega(2^n)$
On the Boolean hypercube, the volume of a Hamming ball with radius $r < n/2$ is exponentially small. As n increases, the distance between any two points concentrates around $n/2$ with variance $\Theta(\sqrt{n})$. The "signal" from the first bit (a distance change of 4) is drowned out by the noise of the irrelevant dimensions, requiring an exponential number of samples to find a neighbor close enough to preserve the parity of x_1.

Comparative Discussion: Winnow is the clear winner for this sparse learning task because it effectively "ignores" irrelevant dimensions via multiplicative updates. Perceptron and Least Squares are both sensitive to the total dimension n, resulting in linear scaling. 1-NN is fundamentally unsuited for this high-dimensional problem, as it fails to capture the sparse structure and suffers from super-exponential sample requirements relative to the error threshold.

(d) Analytical Derivation of the Perceptron Mistake Probability Bound

To derive a non-trivial upper bound $\hat{p}_{m,n}$ on the probability that the perceptron makes a mistake on a randomly selected s-th example, we apply the **Perceptron**

Mistake Bound to the specific geometry of the "just a little bit" problem.

1. Problem Geometry and Parameters

The target concept is defined by $y = x_1$, which implies a target weight vector $\mathbf{w}^* = (1, 0, \ldots, 0)^T$.

- **Margin (γ):** For any pattern $\mathbf{x} \in \{-1, 1\}^n$, the margin γ is given by $y(\mathbf{w}^* \cdot \mathbf{x}) = x_1(x_1) = x_1^2 = 1$. Hence, the geometric margin is exactly $\gamma = 1$.
- **Radius (R):** The patterns are sampled from the Boolean hypercube $\{-1, 1\}^n$. The squared norm of any point $\mathbf{x}$ is $\|\mathbf{x}\|^2 = \sum_{j=1}^{n} x_j^2 = n$. Thus, the radius is $R = \sqrt{n}$.

2. Total Mistake Bound

For any sequence of data separable by a margin γ and bounded by radius R, the total number of mistakes M made by the Perceptron algorithm is bounded by:

$$M \leq \frac{R^2}{\gamma^2}$$

Substituting the values for this problem:

$$M \leq \frac{n}{1^2} = n$$

This result demonstrates that the Perceptron will make at most n mistakes throughout the entire training process, regardless of the total number of examples m.

Note also that the number of mistakes by the Perceptron algorithm cannot exceed the total number of examples, since the perceptron can at most make one mistake per trial. Therefore:

$$M \leq min(m, n)$$

3. Derivation of the Probability Bound $\hat{p}_{m,n}$

Let M_m denote the total number of mistakes made after processing m examples. From the bound above, we have $M_m \leq n$.

If we sample an integer $s \in \{1, \ldots, m\}$ uniformly at random, and considering that the sequence S is drawn from D^m, the probability that the Perceptron makes a mistake on the s-th example is given by the expectation over the distribution and the random index:

$$\hat{p}_{m,n} = P_{S \sim D^m,\, s \sim [m]}(\text{mistake on } s) = \frac{E[M_m]}{m}.$$

Applying the upper bound $M_m \leq min(n, m)$:

$$\hat{p}_{m,n} \leq min(1, \frac{n}{m})$$

Note on Generalization: While this bound $\hat{p}_{m,n}$ applies to the average mistake probability within the sequence, a standard leave-one-out analysis for the generalization error on a future $(m+1)$-th example would similarly yield the bound

$$\hat{p} \leq \frac{n}{m+1}.$$

For this specific problem where $s \in \{1, \ldots, m\}$, we utilize m as the denominator.

Conclusion

The derived non-trivial upper bound is $\hat{\mathbf{p}}_{\mathbf{m},\mathbf{n}} = \frac{\mathbf{n}}{\mathbf{m}}$. This bound is independent of s and implies that for a fixed dimension n, the probability of a mistake vanishes as the number of examples m increases.

(e) 1-NN Sample Complexity Lower Bound

The "just a little bit" problem illustrates the extreme sensitivity of the 1-Nearest Neighbor (1-NN) algorithm to the curse of dimensionality. The task is to predict the first bit of a Boolean vector $x \in \{-1, 1\}^n$ based on a training set of m points. To achieve a generalization error of at most 10%, the nearest neighbor x' of a test point x must satisfy

$$P(x'_1 = x_1) \geq 0.9.$$

Distance decomposition. Consider the squared Euclidean distance between two points x and x':

$$\|x - x'\|_2^2 = (x_1 - x_1')^2 + \sum_{i=2}^{n} (x_i - x_i')^2.$$

- The first term $(x_1 - x_1')^2$ represents the *signal* from the relevant bit. It equals 0 if the first bits match and 4 if they differ. - The sum over $i = 2, \ldots, n$ represents the *noise* from the remaining $n - 1$ irrelevant bits. Each term is independently 0 or 4 with probability 1/2 for a random training point.

Let

$$D_{\text{noise}} = \sum_{i=2}^{n} (x_i - x_i')^2$$

denote the noise contribution. Its mean and standard deviation are

$$E[D_{\text{noise}}] = 2(n-1), \qquad \sigma = \sqrt{\text{Var}(D_{\text{noise}})} = \Theta(\sqrt{n}).$$

Signal vs. noise. The signal contribution from the first bit is at most 4, whereas the typical fluctuation of the noise is $\Theta(\sqrt{n})$. As n grows, the noise dominates the signal, making the difference between neighbors with correct or incorrect first bits almost indistinguishable.

Nearest-neighbor distance in high dimensions. For a training set of size m, the closest neighbor in the noise subspace concentrates around

$$D_{\text{noise}}^{\min} \approx 2(n-1) - \Theta(\sqrt{n \log m}),$$

by standard concentration of measure arguments. This shows that even the nearest neighbor among m random points will still have a large noise distance.

To ensure that the signal from the first bit affects the nearest-neighbor decision, the nearest neighbor must be *atypically close* in the noise subspace, requiring

$$\sqrt{n \log m} = \Theta(n) \quad \Longrightarrow \quad \log m = \Theta(n) \quad \Longrightarrow \quad m = 2^{\Theta(n)}.$$

Conclusion. Any polynomial-sized training set in n is insufficient for 1-NN to exploit the relevant feature, and the classification error approaches 1/2. To maintain the error below a fixed constant $\epsilon < 1/2$, the required sample complexity for 1-NN satisfies

$$\boxed{m = \Omega(2^n)}.$$

This demonstrates that 1-NN is fundamentally unsuited to this sparse, high-dimensional task, in contrast to algorithms like Winnow, which exploit sparsity and achieve logarithmic sample complexity in n.