COMP0078 Assignment 2

Student Numbers: 21168615 & 19004608

Dec 14, 2022

1 PART I

1.1 Kernel Perceptron

1.1.1 Experimental Results

1. To generalise the kernel perceptron to K classes:

Algorithm 1 An training algorithm for multi-class kernel perceptron

```
for k ∈ {1, 2, ..., K} do                                ▷ Initialise weights for first training example
    w_k^{1,1} ← 0
end for
for i ∈ {1, 2, ..., M} do:                               ▷ Number of Epochs
    if i==1 then                                         ▷ Skip first data point if it's the first epoch
        j_init ← 2
    else
        j_init ← 1
    end if
    for j ∈ {j_init, ..., N} do                          ▷ Number of training points
        for k ∈ {1, 2, ..., K} do                        ▷ Number of classes
            ŷ_k^{i,j} ← Σ_{i'=1}^{i-1} Σ_{j'=1}^{j-1} (w_k^{i',j'} · k(x_{j'}, x_j))   ▷ Prediction for class k with kernel k(·,·)
            if sign(ŷ_k^{i,j}) ≠ sign(y_k^{i,j}) then    ▷ Compare predicted ŷ_k^{i,j} and actual y_k^{i,j})
                w_k^{i,j} ← y_k^{i,j}
            else
                w_k^{i,j} ← 0
            end if
        end for
    end for
end for
```

Note that $y_k^{i,j}$ is the one-hot encoding of the j^{th} data point in the i^{th} epoch for the k^{th} class.

The number of epochs was determined by training on the mini training data set until the performance of the mini testing data set no longer changed. This was considered a reasonable number of epochs to use to train with the actual data set.

For prediction, the k with the maximum value of $\hat{y}_k$ (the argmax of $\hat{y}$) is considered the label prediction by the model.

Basic Results:

	Number of Training Epochs	Train Error	Test Error
degree=1.0e+00	5	8.59%±1.10%	10.63%±1.21%
degree=2.0e+00	2	2.88%±0.58%	5.69%±0.62%
degree=3.0e+00	1	2.78%±0.72%	5.29%±0.80%
degree=4.0e+00	3	0.32%±0.14%	3.27%±0.44%
degree=5.0e+00	1	1.86%±0.38%	4.70%±0.59%
degree=6.0e+00	2	0.32%±0.07%	3.24%±0.36%
degree=7.0e+00	2	0.27%±0.08%	3.32%±0.45%

Figure 1: 20 Runs with a Polynomial Kernel

2. Cross Validation Results:

Run	Optimal degree	Number of Training Epochs	Train Error	Test Error
1	4.0e+00	3	0.31%	3.28%
2	4.0e+00	3	0.38%	3.28%
3	6.0e+00	2	0.03%	3.28%
4	7.0e+00	2	0.09%	2.80%
5	4.0e+00	3	0.13%	2.96%
6	4.0e+00	3	0.38%	2.96%
7	6.0e+00	2	0.11%	2.58%
8	4.0e+00	3	0.30%	3.28%
9	4.0e+00	3	0.31%	3.01%
10	4.0e+00	3	0.20%	3.60%
11	4.0e+00	3	0.26%	3.87%
12	6.0e+00	2	0.12%	2.90%
13	6.0e+00	2	0.16%	3.17%
14	6.0e+00	2	0.19%	2.53%
15	4.0e+00	3	0.22%	2.47%
16	4.0e+00	3	0.26%	4.03%
17	6.0e+00	2	0.07%	2.69%
18	4.0e+00	3	0.35%	3.82%
19	4.0e+00	3	0.38%	2.26%
20	7.0e+00	2	0.05%	3.82%
Across Runs	4.9e+00±1.1e+00	2.60±0.49	0.21%±0.11%	3.13%±0.50%

Figure 2: 5-fold Cross-Validation for 20 Runs with a Polynomial Kernel

3. Confusion Matrix:

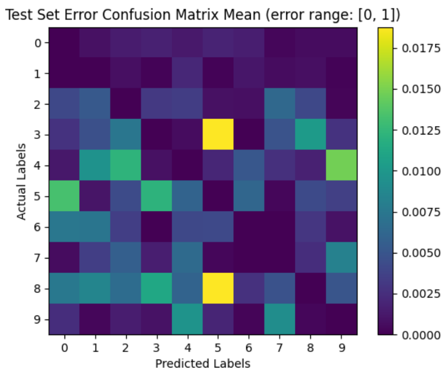

Figure 3: Polynomial Kernel Mean

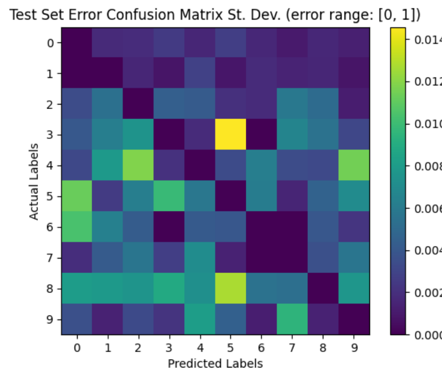

Figure 4: Polynomial Kernel St. Dev.

		Predicted Labels									
		0	1	2	3	4	5	6	7	8	9
Actual Labels	0	0.0%±0.0%	0.08%±0.18%	0.14%±0.19%	0.17%±0.25%	0.13%±0.16%	0.19%±0.27%	0.16%±0.16%	0.04%±0.11%	0.06%±0.16%	0.06%±0.13%
	1	0.0%±0.0%	0.0%±0.0%	0.08%±0.16%	0.02%±0.08%	0.21%±0.28%	0.02%±0.09%	0.12%±0.18%	0.06%±0.14%	0.08%±0.16%	0.02%±0.08%
	2	0.4%±0.34%	0.52%±0.54%	0.0%±0.0%	0.31%±0.44%	0.34%±0.42%	0.08%±0.2%	0.08%±0.18%	0.63%±0.58%	0.41%±0.51%	0.03%±0.12%
	3	0.27%±0.4%	0.46%±0.62%	0.74%±0.75%	0.0%±0.0%	0.06%±0.18%	1.87%±1.46%	0.0%±0.0%	0.48%±0.66%	1.02%±0.55%	0.28%±0.31%
	4	0.12%±0.32%	0.96%±0.79%	1.22%±1.18%	0.09%±0.21%	0.0%±0.0%	0.2%±0.33%	0.51%±0.62%	0.26%±0.34%	0.17%±0.32%	1.48%±1.15%
	5	1.33%±1.13%	0.11%±0.25%	0.43%±0.62%	1.21%±0.99%	0.6%±0.58%	0.0%±0.0%	0.62%±0.61%	0.03%±0.15%	0.42%±0.46%	0.35%±0.7%
	6	0.74%±1.04%	0.73%±0.64%	0.35%±0.43%	0.0%±0.0%	0.41%±0.41%	0.41%±0.39%	0.0%±0.0%	0.0%±0.0%	0.29%±0.4%	0.09%±0.22%
	7	0.06%±0.19%	0.34%±0.42%	0.57%±0.57%	0.15%±0.27%	0.65%±0.71%	0.03%±0.14%	0.0%±0.0%	0.0%±0.0%	0.25%±0.36%	0.82%±0.57%
	8	0.75%±0.8%	0.86%±0.79%	0.67%±0.75%	1.12%±0.89%	0.6%±0.72%	1.88%±1.27%	0.26%±0.55%	0.47%±0.53%	0.0%±0.0%	0.5%±0.76%
	9	0.24%±0.36%	0.03%±0.14%	0.15%±0.33%	0.09%±0.22%	0.96%±0.81%	0.19%±0.44%	0.03%±0.12%	0.92%±0.95%	0.03%±0.14%	0.0%±0.0%

Figure 5: Polynomial Kernel Confusion Matrix Error Rate Table

4. The images with the most errors made by the 20 models trained in part 2 were considered the hardest to predict and visualised. It is not surprising that these are hard to predict. We can see that in all five cases, the images do not look like their labels, in fact, they mostly look like vertical or horizontal bars. Thus it would be unreasonable to expect a model to predict them correctly.

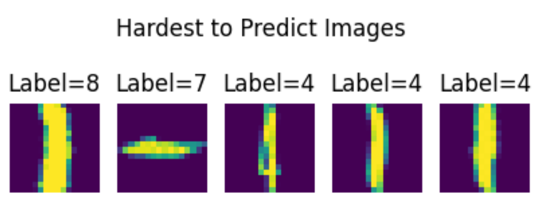

Figure 6: Images with the Most Errors from the Trained Models

5. To select a good range of values for the kernel width sigma for the Gaussian kernel, the mini training and testing sets was used to quickly gauge the test error for different hyper-parameters. The hyper-parameter values above and below the best performing hyper-parameter (with respect to test set error) was chosen as the hyper-parameter range to define S, the set of values for cross-validation when repeating parts 1 and 2 with the Gaussian Kernel.

	Number of Training Epochs	Train Error	Test Error
sigma=1.0e-05	6	22.44%±4.69%	22.74%±6.04%
sigma=2.1e-05	4	11.24%±2.05%	11.94%±2.08%
sigma=4.3e-05	5	8.54%±1.79%	9.24%±1.89%
sigma=8.9e-05	11	2.65%±0.96%	4.20%±1.63%
sigma=1.8e-04	5	2.78%±0.84%	4.11%±1.54%
sigma=3.8e-04	10	2.62%±1.66%	4.04%±2.14%
sigma=7.8e-04	3	2.04%±0.95%	2.83%±1.55%
sigma=1.6e-03	5	0.56%±0.34%	2.36%±1.37%
sigma=3.4e-03	3	0.28%±0.33%	2.23%±1.00%
sigma=7.0e-03	2	0.17%±0.20%	1.91%±0.99%
sigma=1.4e-02	2	0.10%±0.15%	1.72%±0.99%
sigma=3.0e-02	2	0.03%±0.06%	1.59%±1.02%
sigma=6.2e-02	1	1.44%±0.43%	3.41%±1.36%
sigma=1.3e-01	2	0.06%±0.11%	3.60%±1.63%
sigma=2.6e-01	2	0.12%±0.14%	3.73%±1.48%
sigma=5.5e-01	1	0.62%±0.31%	6.40%±1.22%
sigma=1.1e+00	1	0.09%±0.08%	36.11%±3.68%
sigma=2.3e+00	1	0.15%±0.03%	57.32%±3.65%
sigma=4.8e+00	1	0.15%±0.03%	60.54%±3.61%
sigma=1.0e+01	1	0.15%±0.03%	60.70%±3.52%

Figure 7: Performance for mini data set (to narrow down a hyper-parameter range)

Basic Results:

	Number of Training Epochs	Train Error	Test Error
sigma=1.4e-02	2	0.31%±0.10%	3.37%±0.37%
sigma=2.2e-02	3	0.04%±0.02%	2.96%±0.31%
sigma=3.0e-02	2	0.18%±0.06%	3.28%±0.27%
sigma=3.8e-02	1	1.38%±0.17%	4.68%±0.45%
sigma=4.6e-02	3	0.05%±0.04%	3.75%±0.35%
sigma=5.4e-02	3	0.05%±0.03%	4.06%±0.44%
sigma=6.2e-02	1	1.77%±0.15%	5.61%±0.45%

Figure 8: 20 Runs with a Gaussian Kernel

Cross Validation Results:

Run	Optimal sigma	Number of Training Epochs	Train Error	Test Error
1	2.2e-02	3	0.05%	2.69%
2	2.2e-02	3	0.07%	2.63%
3	2.2e-02	3	0.04%	3.66%
4	2.2e-02	3	0.01%	2.69%
5	2.2e-02	3	0.04%	2.10%
6	2.2e-02	3	0.04%	2.37%
7	2.2e-02	3	0.07%	2.85%
8	2.2e-02	3	0.07%	2.69%
9	2.2e-02	3	0.05%	2.42%
10	2.2e-02	3	0.01%	2.47%
11	2.2e-02	3	0.00%	2.31%
12	2.2e-02	3	0.04%	2.90%
13	2.2e-02	3	0.08%	2.63%
14	2.2e-02	3	0.05%	2.90%
15	2.2e-02	3	0.05%	1.88%
16	2.2e-02	3	0.01%	2.63%
17	2.2e-02	3	0.08%	1.99%
18	2.2e-02	3	0.04%	2.85%
19	2.2e-02	3	0.04%	2.42%
20	2.2e-02	3	0.13%	3.49%
Across Runs	2.2e-02±3.5e-18	3.00±0.00	0.05%±0.03%	2.63%±0.42%

Figure 9: 5-fold Cross-Validation for 20 Runs with a Gaussian Kernel

6. An alternative method to generalise the kernel perceptron to k-classes:

Algorithm 2 Another training algorithm for multi-class kernel perceptron

for $k \in \{1, 2, ..., K\}$ **do** ▹ Initialise weights for first training example
  $w_k^{1,1} \leftarrow 0$
end for
for $i \in \{1, 2, ..., M\}$ **do**: ▹ Number of Epochs
  if i==1 **then** ▹ Skip first data point if it's the first epoch
    $j_{init} \leftarrow 2$
  else
    $j_{init} \leftarrow 1$
  end if
  for $j \in \{j_{init}, ..., N\}$ **do** ▹ Number of training points
    for $k \in \{1, 2, ..., K\}$ **do** ▹ Number of classes
      $\hat{y}_k^{i,j} \leftarrow \sum_{i'=1}^{i-1} \sum_{j'=1}^{j-1} (w_k^{i',j} \cdot k(\mathbf{x}_{j'}, \mathbf{x}_j))$ ▹ Prediction for class k with kernel $k(\cdot, \cdot)$
    end for
    $\hat{c} \leftarrow argmax_k \hat{y}_k^{i,j}$ ▹ Predicted label
    $c \leftarrow argmax_k y_k^{i,j}$ ▹ Actual label
    if $\hat{c} \neq c$ **then** ▹ Compare predicted and actual label
      $w_c^{i,j} \leftarrow y_c^{i,j}$ ▹ Non-zero weights only for actual and predicted label
      $w_{\hat{c}}^{i,j} \leftarrow y_{\hat{c}}^{i,j}$
      $w_k^{i,j} \leftarrow 0 \forall k \in \{1, 2, ..., K\}, k \neq c, k \neq \hat{c}$
    else ▹ All zero weights if actual and predicted labels match
      $w_k^{i,j} \leftarrow 0 \forall k \in \{1, 2, ..., K\}$
    end if
  end for
end for

Basic Results:

	Number of Training Epochs	Train Error	Test Error
degree=1.0e+00	5	9.47%±2.42%	11.66%±2.36%
degree=2.0e+00	3	3.46%±1.15%	6.44%±1.29%
degree=3.0e+00	2	2.33%±0.66%	5.35%±0.69%
degree=4.0e+00	2	1.52%±0.45%	4.56%±0.71%
degree=5.0e+00	2	0.98%±0.34%	4.10%±0.39%
degree=6.0e+00	2	0.62%±0.14%	3.68%±0.44%
degree=7.0e+00	2	0.59%±0.20%	3.79%±0.49%

Figure 10: 20 Runs with a Polynomial Kernel (Alternative Training Method)

Cross Validation Results:

Run	Optimal degree	Number of Training Epochs	Train Error	Test Error
1	7.0e+00	2	0.69%	3.60%
2	6.0e+00	2	0.62%	3.66%
3	6.0e+00	2	0.75%	4.19%
4	5.0e+00	2	0.99%	4.30%
5	7.0e+00	2	0.77%	3.87%
6	7.0e+00	2	0.77%	4.09%
7	5.0e+00	2	0.95%	4.03%
8	7.0e+00	2	0.55%	3.87%
9	7.0e+00	2	0.50%	3.98%
10	6.0e+00	2	0.93%	3.92%
11	6.0e+00	2	0.59%	3.71%
12	7.0e+00	2	0.62%	4.19%
13	6.0e+00	2	0.67%	4.14%
14	7.0e+00	2	0.63%	3.71%
15	6.0e+00	2	0.67%	3.60%
16	6.0e+00	2	0.95%	3.82%
17	7.0e+00	2	0.65%	3.17%
18	7.0e+00	2	0.66%	4.57%
19	5.0e+00	2	0.78%	3.92%
20	6.0e+00	2	1.26%	5.05%
Across Runs	6.3e+00±7.1e-01	2.00±0.00	0.75%±0.18%	3.97%±0.39%

Figure 11: 5-fold Cross-Validation for 20 Runs with a Polynomial Kernel (Alternative Training Method)

1.1.2 Discussions

For these experiments, the number of folds, number of runs, and train/test split were parameters that were arbitrarily chosen, not chosen through cross-validation. Moreover, the kernel parameters were cross-validated to find optimal values, however the range for which cross validation was performed over (i.e. degree 1 to 7 for the polynomial kernel) was chosen relatively arbitrarily. For Gaussian kernels, this was done in a two step process, where a very wide range for sigma was initially used, although this wide range was still chosen arbitrarily. Increasing the number of folds would provide a better estimate of the out of sample performance (i.e. in the limit where we have N folds, each training on $N-1$ points and testing on the single excluded data point). However, this can be quite computationally heavy and a compromise must be drawn. This a similar situation for the number of runs. By increasing the number of runs, we gain a better understanding of average performance, however this is at a computational cost of performing more runs. The train/test split used for these experiments was 80% training data and 20% testing data. This follows general rule of thumb for train/test splits. Given a finite data set size, increasing the training percentage can provide inaccurate out of sample performance with the small test set. Conversely, a smaller training percentage can poorly represent the model's performance due to limited training examples. Thus a balance must be drawn between the two. The range of hyper-parameters to search for cross validation requires past experience working with kernels and knowledge about the given data set. However these ranges can be quite general (as shown for the Gaussian kernel where we search a space from $1e-05$ to $1e+01$) and we can iteratively narrow our search space.

Comparing the two methodologies for generalising the perceptron to k-class classifiers, we can see that the main implementation difference is with the training process, where Algorithm 1 can update an arbitrary number of its weights to be non-zero while Algorithm 2 will update only two weights if the prediction is incorrect. In this sense, Algorithm 1 is like training k independent binary classifiers, where updates are made for each sub-classifier independent of other the prediction of the others. Algorithm 2 introduces dependence between sub-classifiers during the training phase. This is because whether a sub-classifier is selected to be updated depends on if it is the label predicting classifier (i.e. the argmax) or the classifier for the actual label. Otherwise, it is not updated. In some sense, Algorithm 1 is able to pass "correction" signal to all of its classifiers at each step, while Algorithm 2 is constrained in its "speed" of training its classifiers to a maximum of two at a time. Comparing Figure 2 and Figure 11, we can see that Algorithm 1 performs better than Algorithm 2 and this is likely due to this ability of Algorithm 1 to perform corrections for all its classifiers even if they are not the actual label or wrongly predicted label for a training example.

Comparing Figure 2 and Figure 9, we can see that the Gaussian Kernel performs better than the Polynomial Kernel. This is expected because the Gaussian Kernel provides a more expressive (infinite) feature space to model the data compared to a Polynomial kernel.

These experiments were implemented by vectorising and simulating the training behaviour of this online algorithm for faster computation. However, the results exactly mimic the behaviour of the algorithms depicted. The vectorisation starts with initialising a zeros weight matrix $\mathbf{w} = \mathbf{0} \in \mathbb{R}^{KxN}$. During the j^{th} step of training, we only update the j^{th} column of weights in $\mathbf{w}$. Thus, during prediction for the j^{th} training example, we know that all weights for training examples $\geq j$ will be zero and we will be simulating an online algorithm that only predicts using the $< j$ examples. Moreover during prediction, the operation $\sum_{i'=1}^{i-1}\sum_{j'=1}^{j-1}(w_k^{i',j'} \cdot k(\mathbf{x}_{j'}, \mathbf{x}_j)$ can be factored as $\sum_{j'=1}^{j-1}(\sum_{i'=1}^{i-1} w_k^{i',j'}) \cdot k(\mathbf{x}_{j'}, \mathbf{x}_j)$. Thus, instead of storing a weight matrix of size $\mathbb{R}^{Kx(N \cdot M)}$, for M epochs, we can sum the weights corresponding to the sample training example over multiple epochs and have a weight matrix of constant size $\mathbb{R}^{KxN}$. The kernel evaluations were also precomputed to simulate the online model and speed up computation. Precomputing the gram matrix for a kernel helps vectorise the operation. Thus during the simulated online model, the gram matrix is simply indexed at the appropriate training example to perform the weight updates. Additionally, across different hyper-parameters for a given kernels, a ”pre-gram matrix” is pre-computed and used across the different hyper-parameters of a kernel. For example with a polynomial kernel, the gram matrix of inner products is computed as a ”pre-gram matrix”. Then for each polynomial experiment, the elements of the matrix is simply raised to the d^{th} degree. This also significantly improves the speed of the experiments because a single ”pre-gram matrix” can be shared across experiments during a hyper-parameter search. Although the perceptron training algorithm is sequential in nature (i.e. the new weight update depends on the weight updates of the previous step), each hyper-parameter and each run is independent. This invites the opportunity to run these in parallel, performing weight updates for multiple runs in parallel. However we ran into issues of memory constraints when trying to run this on our machine with the given data set, so our implementation did not take advantage of this.

2 PART II

2.1 Semi-supervised Learning via Laplacian Interpolation

2.1.1 Experimental results

		# of known labels per class				
		1	2	4	8	16
# of datapoints per class	50	0.16±0.099	0.12±0.1041	0.08±0.0644	0.04±0.013	0.04±0.0229
	100	0.11±0.1253	0.07±0.0213	0.06±0.0279	0.06±0.0234	0.04±0.0219
	200	0.09±0.0996	0.05±0.0387	0.02±0.0086	0.02±0.0127	0.02±0.0104
	400	0.08±0.1437	0.02±0.0071	0.02±0.0063	0.02±0.0063	0.02±0.0038

Figure 12: Laplacian Interpolation

And for the laplacian kernel method:

		# of known labels per class				
		1	2	4	8	16
# of datapoints per class	50	0.1±0.0844	0.1±0.1211	0.05±0.017	0.04±0.015	0.05±0.0189
	100	0.13±0.1813	0.07±0.0106	0.06±0.0176	0.05±0.0181	0.05±0.023
	200	0.02±0.0376	0.02±0.0063	0.02±0.0074	0.02±0.0057	0.02±0.0083
	400	0.02±0.0036	0.01±0.0052	0.02±0.0048	0.01±0.0052	0.02±0.0043

Figure 13: Laplacian Kernel Interpolation

2.1.2 Discussions

2. Some initial observations include that both models exhibit relatively high accuracy in all low-data settings. Additionally, both error and variance are observed to decrease as a function of both number of samples and number of total datapoints. Further, we observe that the kernel interpolation approach gives lower error and variance across all experiments, and also is less severely affected by sample number.

3. To analyse this difference in model performance, we first perform a visualisation of our induced 3-NN graph seen in **Fig 14**: note that blue edges are where both labels are -1, teal when both are different, and yellow when both are +1. From our diagram we can observe that the graph is highly separable into 2 distinct clusters, and these clusters strongly align with label predictions.

   A reasonable explanation for the kernel approach's superior performance follows from the observation that seminorm interpolation propagates local information to pass labels through the graph, whereas the kernel method utilises global graph connectivity properties of each point to a set of labels to make predictions. This means that in the seminorm method, only datapoints close to labelled data receive information from the labels themselves, and accuracy is likely reduced for datapoints far from the labels as label information is passed indirectly through the graph. Further if a point is close the boundary of a cluster, it may be be misclassified due to its proximity to a single label from the other cluster. In contrast, the kernel interpolation method takes global information from all the labelled datapoints and weights them according to their connectedness through the graph to a given vertex to determine a predicted label. A vertex will only be given a label if it is more strongly connected to all labelled points that have this label. This use of global information helps to resolve the case in which a vertex lies on the boundary of the two clusters as the model draws conclusions directly from its connectedness to labelled points rather than receiving information from similarly ambiguous boundary points. However, we should note that this superior performance is an artifact of the dataset, and that it separates well into one cluster per datapoint: this implies that if a vertex is more strongly connected to points with label +1, then it is most likely given this label in reality.

4. We may postulate that the seminorm interpolation method may outperform the kernel method on datasets where global information is more ambiguous than local information: for instance, consider a graph where there are more than 2 clusters, each with coherent binary labelling, but perhaps where 2 clusters of the same label are separated by a cluster of different labelling. In this scenario, the seminorm method would label the data in 3 clusters, by only considering local edgewise label changes, but the kernel method may have difficulty correctly identifying the label for some vertices as it may be less connected to points of the same label in ground truth, than to points of a different label in the neighbouring cluster.

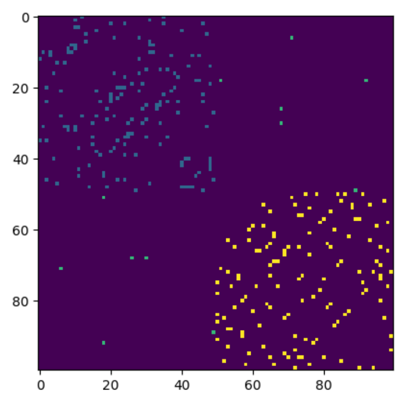

Figure 14: Graph adjacency matrix for n = 50 datapoints per label

3 PART III

3.1 Questions

a. Sample complexity plots for the four algorithms: perceptron, winnow, least squares, and 1-nearest neighbours.

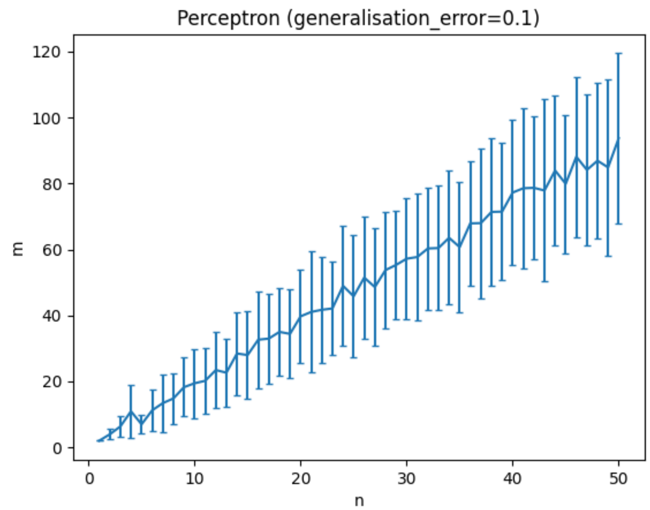

Figure 15: Perceptron Sample Complexity

Figure 16: Winnow Sample Complexity

‘

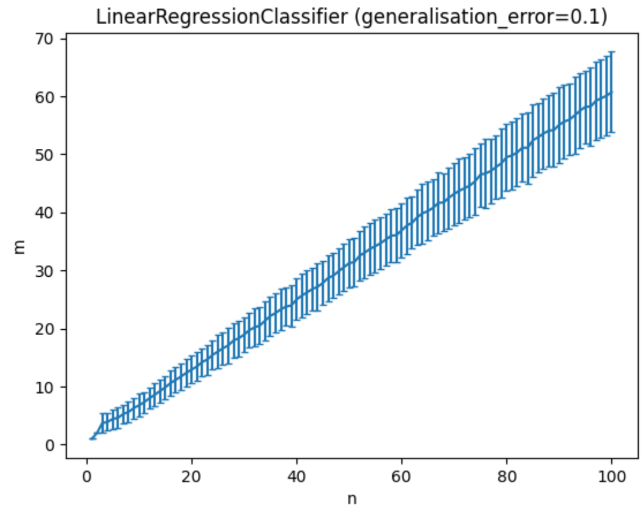

Figure 17: Least Squares Sample Complexity

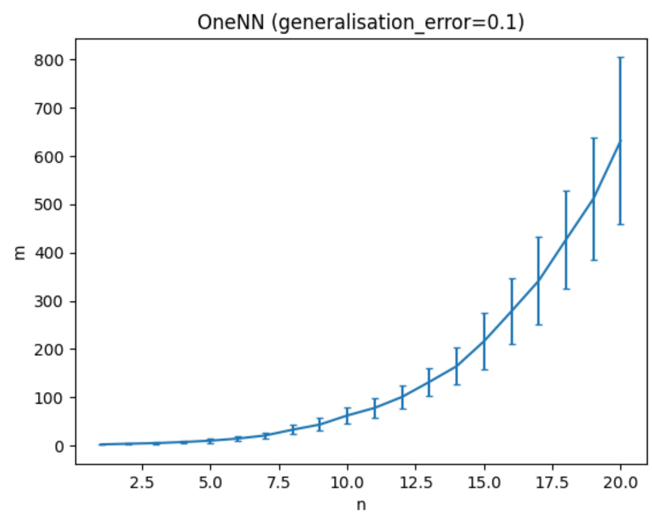

Figure 18: 1-Nearest Neighbours Sample Complexity

b. The following algorithm was used to estimate sample complexity:

Algorithm 3 Sample Complexity Estimation Algorithm

```
for n ∈ {1,2,...,N} do:                                        ▷ Number of dimensions
    M_n ← []                                                   ▷ Store m values for each trial
    for t ∈ {1,2,...,T} do                                     ▷ Number of trials
        m ← 0                                                  ▷ Number of training points
        D_train ← {}                                           ▷ Training set
        D_test ← {(x^1_test, y^1_test), ..., (x^M'_test, y^M'_test)}   ▷ Fixed test set of size M'
        error ← 1                                              ▷ initial test set error
        while error > 0.1 do
            m ← m + 1
            D_train ← D_train ∪ {(x^m_train, y^m_train)}       ▷ Add a new training point
            error ← model.train(D_train).test(D_test)          ▷ Train model and compute error
        end while
        M_n ← M_n + [m]
    end for
```

For each $\mathcal{M}_n$ the mean and standard deviation were calculated to produce the plots for each model. Choosing different values for the number of trials T and the test set size M', is a tradeoff between limited computation resources and accuracy of our approximation. With larger number of trials, we improve our approximation, however this requires more computational resources. A larger test set size M' provides a better approximation of $\{-1, 1\}^n$ and therefore a better approximation of the generalisation error. Moreover, choosing N puts a finite horizon on the number of points available for estimating the sample complexity function. For some algorithms such as one-nearest neighbours, we see that the number of training samples grows exponentially with n. Thus to have a plot up to 100 dimensions like with least squares, would require a lot more computational resources. As such, we limited our evaluation for one-NN to 20 dimensions. However, this can introduce biases in our estimation of the sample complexity. It is possible that our chosen finite horizon is inadequate for capturing the dynamics as $n \to \infty$. For example, a sample complexity that is actually quadratic could seem linear if we do not choose an N that is large enough to reveal this trend.

c. To estimate how m grows as a function of n as $n \longrightarrow \infty$ for each of the four algorithms, we first visualise lines of best fit of the sample complexities for different function classes (i.e. polynomial, exponential, logarithm):

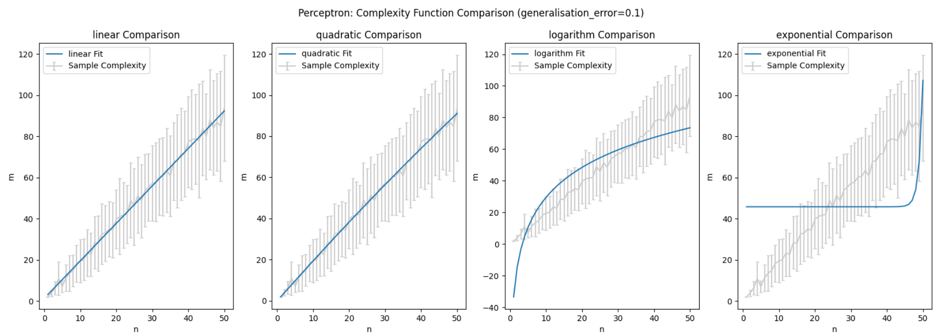

Figure 19: Perceptron Sample Complexity vs Function Classes

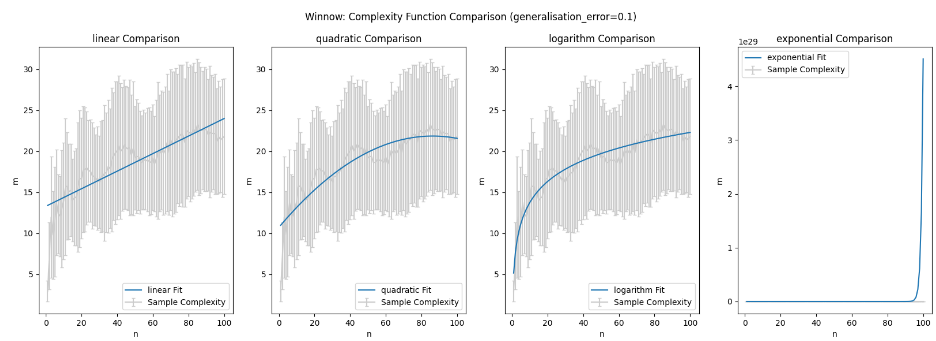

Figure 20: Winnow Sample Complexity vs Function Classes

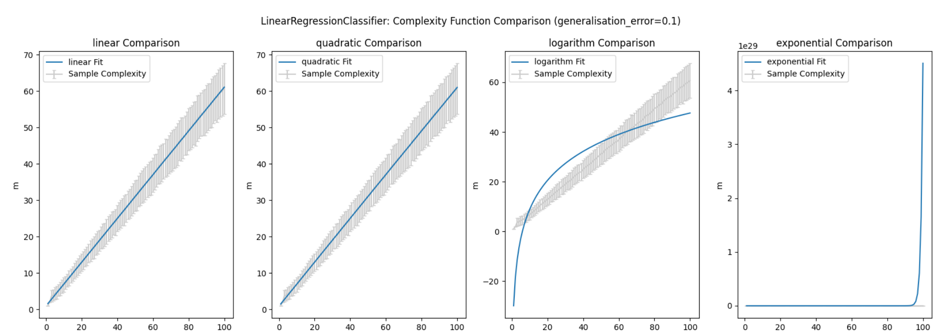

Figure 21: Least Squares Sample Complexity vs Function Classes

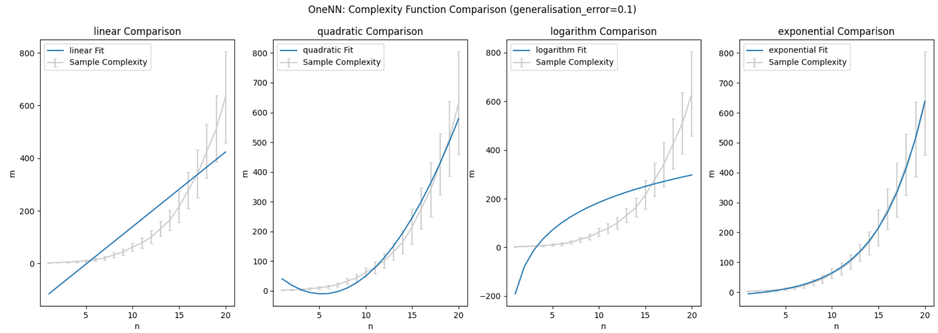

Figure 22: 1-Nearest Neighbours Sample Complexity vs Function Classes

From these plots we may observe:

Perceptron sample complexity grows linearly ($m = \Theta(n)$), and the Winnow sample complexity grows logarithmically ($m = \Theta(\log(n))$). The corresponding asymptotic lower bound results will be proven analytically in the next section.

The least squares sample complexity grows linearly ($m = \Theta(n)$). We may also prove this analytically: note that a linear regression classifier is exactly the same as the 'half space' model shown on page 48 of the learning theory notes, with dimension n. Hence we determine that the VC dimension of the least squares classifier is n. Hence, the fundamental theorem of statistical learning determines that for fixed ϵ, δ, the sample complexity of our least squares classifier is bounded above and below by expressions linear in n.

Finally, the 1-nearest neighbours sample complexity grows exponentially ($m = \Theta(e^n)$).

These estimates can be qualitatively observed by attempting to fit different function curves to the experimental results and choosing the function with the best qualitative fit. This shows that winnow is a much more robust algorithm in high dimensions due to its logarithmic scaling of the number of training points with respect to the dimensionality to achieve the same error rate. On the other hand, one-nearest neighbours scales exponentially, suggesting that it is impractical to use in high dimensional data. This can be attributed to the curse of dimensionality. Both the perceptron and least squares scale linearly, thus performing worse than winnow but better than the one-NN algorithm.

d. We note that since for any X_t , $y_t = X_{1,t}$, we have the following expression representing the linear separability of our dataset: $(v \cdot x_t)y_t \geq 1$ Where $v = (1, 0, \ldots, 0)^T$. Note that $\|v\| = 1$. Hence, in our online mistake bound for the perceptron, we have that our margin parameter $\gamma = 1$. Further note that $\forall t,\ x_t \in \{-1, 1\}^n \implies \|x_t\|^2 = n$. This gives us that $R = max_t \|x_t\| = \sqrt{n}$. Hence our mistake bound for the perceptron algorithm is given by:

$M \leq n.$

The theorem on page 60 of the online learning notes relates mistake bounds to batch generalisation error bounds:

$Prob(\mathcal{A}_S(x') \neq y') \leq \frac{n}{m}$

Corollary

This result gives us a lower bound on sample complexity for the perceptron algorithm:

Suppose we require out generalisation error $Prob(\mathcal{A}_S(x') \neq y') \leq \frac{n}{m}$ to be less than some fixed ϵ.

$\implies \frac{n}{m} \leq \epsilon \implies m \geq \frac{n}{\epsilon}$

Note that given the mistake bound on winnow given by $M \leq 3(log(n) + 1) + 2$ (note k=1 since the ground truth is a 1-literal conjunction given by x_1), following the same approach we arrive at a lower bound on sample complexity for winnow that matches our experimental observations:

$m(n) \geq \frac{3(log(n)+1)+2}{\epsilon} = \Omega(log(n))$

e. From our experimental observations we may expect the sample complexity of 1-NN to be lower bounded by some exponential function. We formalise this in the following proposition:

Proposition:

Following the data-generating distribution described above, we have that our sample complexity given by:

$m(n) = \Omega(2^n)$

Proof

Suppose we sample a training set $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ uniformly from the set $\{-1, 1\}^n$, with associated labels defined by the rule $y|x = x_1$, and use this training set for inference on an arbitrary test point (x, x_1), sampled uniformly from the same set.

We note the following observation:

Observation

if $x \in S$, then $\mathcal{A}_S(x)) = y$, where y is the true label for x.

To justify this, observe that our dataset represents a realiseable learning problem, and as such, if $(x, y), (x\prime, y\prime)$ are datapoints sampled from our distribution, $x = x\prime \implies y = y\prime$. Trivially, x is the 1-nearest neighbour to itself, so the algorithm makes a correct prediction for any datapoint present in our training set.

Hence, $\mathcal{A}_S$ makes an error on x $\implies x \notin S$.

Hence, the set of all training sets that make an error on x is contained in the set of all training sets not containing x.

Hence, for a given x, $P_S(\mathcal{A}_S(x) \neq y) \leq P_S(x \notin S)$.

Since S is a collection of points sampled iid from our data generating distribution, $P_S(x \notin S) = \prod_{i=1}^{m} P(x_i \neq x) = \prod_i (1 - P(x_i = x)) = (1 - 2^{-n})^m$

We note that our choice of x was arbitrary, and since we sampled x uniformly, we arrive at the following generalisation error bound:

$\mathbb{E}_{S \sim \mathcal{D}^m, x \sim \mathcal{D}}[\mathcal{L}_S(x)] \leq (1 - 2^{-n})^m$ Using the identity $1 - x \leq e^{-x}$ provided frequently in the notes, we simplify this bound to give:

$\mathbb{E}_{S \sim \mathcal{D}^m, x \sim \mathcal{D}}[\mathcal{L}_S(x)] \leq exp(-2^n m)$

Suppose we seek m such that our generalisation error is less than some fixed ϵ.

Hence we require $\mathbb{E}_{S \sim \mathcal{D}^m, x \sim \mathcal{D}}[\mathcal{L}_S(x)] \leq exp(-2^n m) \leq \epsilon$

$\implies -m2^{-n} \leq log(\epsilon)$

Hence, $m \geq -log(\epsilon)2^n = log(\frac{1}{\epsilon})2^n$

$\implies m = \Omega(2^n)$ □