COMP0078
Supervised Learning

Coursework 2

by

Alexandra Maria Proca
20047328
alexandra.proca.20@ucl.ac.uk

January 11, 2021

Department of Computer Science
University College London

Contents

1 PART I

1.1 Kernel Perceptron (Handwritten Digit Classification)

1. In the following implementation, the perceptron algorithm is generalized in two ways. First, it utilizes a kernel function in order to map to higher dimensional spaces and distinguish non-linear distributions. A kernel function computes the dot product of input in a feature space $\phi(\boldsymbol{x})$, based on a particular mapping, allowing for a linear boundary to be found within the higher-dimensional representation. One common function is the polynomial kernel, which is of the form $K_d(\boldsymbol{p}, \boldsymbol{q}) = (\boldsymbol{p} \cdot \boldsymbol{q})^d$ and corresponds to a feature space exponential in d.

   The perceptron algorithm is online, operating on a single data point $(\boldsymbol{x}_t, y_t)$ at a time. Thus, training consists of cycling through the training set iteratively for a specified number of epochs. Online learning is defined by the fact that each prediction is based on the previous data encountered. Thus, in the implemented algorithm, the prediction of $\boldsymbol{x}_t$ in the first epoch only uses the weights $\boldsymbol{\alpha}$ and computed kernel K up to t. Following the first epoch, the entirety of $\boldsymbol{\alpha}$ and K are used in prediction as every point has been encountered.

   In order to generalize the perceptron algorithm, which performs binary classification, to K classes, One-versus-All (OVA) classification is implemented. Given a dataset for K classes, OVA creates K binary training sets. For the kth classifier training set, labels in the original data set (y) corresponding to the kth class are set to 1 and all others are set to -1 (y_k). Thus, for $y = k \rightarrow y_k = 1$ and for $y \neq k \rightarrow y_k = -1$. As suggested by the name, each classifier predicts between one class versus all other classes. The predictions of all K classifiers are then compared and the kth classifier yielding the maximum prediction confidence becomes the final prediction k for the particular input [6]. In this particular kernelized perceptron setting, a weight matrix $A \in \mathbb{R}^{K \times N}$ in the higher-dimensional feature space procured by the kernel matrix is updated according to whether each classifier made a mistake at each point. Thus for each classifier, where y_t represents the modified label $-1, 1$ for classifier k at the tth instance,

$$\texttt{if } \hat{y_t} \neq y_t \texttt{ then } \alpha_t = \alpha_{t-1} - \hat{y_t}$$

   The weight matrix $\boldsymbol{w}(\cdot)$ is represented through separate terms $\boldsymbol{\alpha}$ and $K(\boldsymbol{x}, \boldsymbol{x})$, where $\boldsymbol{\alpha}$ is a $K \times N$ matrix. Thus to evaluate $\boldsymbol{w}(\cdot) = \sum_{i=0}^{m} \alpha_i K(\boldsymbol{x}_i, \cdot)$, the dot product of $\boldsymbol{\alpha}$ and K is taken during prediction. To add new terms during training, being that the algorithm is online, the dot product is taken over the elements of $\boldsymbol{\alpha}$ and K which have been observed. Thus, in the dot product, $\boldsymbol{\alpha}$ and K at time $t+1$ will have an additional column than $\boldsymbol{\alpha}$ and K at time t.

   Basic results are found by performing 20 runs for each parameter $d = 1, \ldots, 7$, the specified degree of the polynomial kernel. For each run, a random train-test split of 4:1 is performed, the perceptron is trained on the training set for 25 epochs, and the test error is computed for the training and test sets. The mean and standard deviation of the train and test error are then computed over the 20 runs for each parameter. From initial experimental observations, the maximum number of epochs to converge over all parameters is found to be 25. Thus, training is performed over 25 epochs to ensure convergence at each value of d, given that the kernel perceptron converges at different rates depending on its parameterization. An alternative approach could be to determine the optimal number of epochs to train to for each d using cross validation, given that training to convergence does not always yield the highest test error as a model may overfit.

Table 1: Basic results for OVA polynomial kernel degree.

d	Train Error	Test Error
1	0.06847±0.00913	0.09847±0.01043
2	0.00032±0.00023	0.03054±0.00487
3	0.00020±0.00021	0.02777±0.00408
4	0.00013±0.00011	0.02610±0.00308
5	0.00012±0.00008	0.02710±0.00358
6	0.00013±0.00010	0.02777±0.00358
7	0.00013±0.00009	0.02825±0.00329

The initial results display a trend of decreasing train and test error up to $d = 4$ and then a trend of increasing test error afterwards. This suggests that $d = 4$ may be optimal, as higher polynomial degrees overfit the data.

2. Cross-validation is a resampling technique that can be used to mimic test data within the train set by leaving a subset of datapoints out as test sets. 5-fold cross-validation is used to split the training data into 5 folds, over which, on the tth run (out of 5 total runs), the tth fold is treated as a test fold and the rest are treated as a train fold. For 20 runs with different random 4:1 train test splits, cross-validation is employed to select the parameter d^* to train and test on with the full dataset. For each parameter, 5-fold cross-validation is used on a particular train split to determine the mean test fold error, which is used to determine d^*. d^* is then used to train on the full train set and test on the test set.

Table 2: OVA polynomial kernel perceptron cross-validation results.

Mean d*	Mean Test Error
3.9±0.83066	0.02769±0.00362

3. A confusion matrix can be used to display the mistakes made by a model by recording the predicted label versus the true label. Thus, for each misclassification made by the perceptron during testing, the corresponding prediction and label is stored in such a matrix and normalized by the number of labels in the test set corresponding to the true label. This representation allows one to identify classes that are more difficult for the model to distinguish. In this implementation, for 20 runs generating different 4:1 train-test splits, the perceptron is trained and a confusion matrix is generated during testing. The mean and standard deviation of the confusion matrices generated are then taken to yield a final average confusion matrix across all runs.

Table 3: Confusion Matrix.

		Predicted Label									
		0	1	2	3	4	5	6	7	8	9
True Label	0	0.0±0.0	0.0±0.001	0.002±0.003	0.001±0.002	0.000±0.001	0.001±0.002	0.002±0.002	0.000±0.001	0.000±0.001	0.001±0.001
	1	0.0±0.001	0.0±0.0	0.0±0.001	0.0±0.001	0.002±0.002	0.0±0.0	0.001±0.002	0.0±0.001	0.001±0.003	0.0±0.001
	2	0.005±0.007	0.002±0.004	0.0±0.0	0.007±0.006	0.006±0.005	0.002±0.003	0.0±0.001	0.006±0.004	0.005±0.005	0.0±0.0
	3	0.001±0.002	0.001±0.003	0.006±0.004	0.0±0.0	0.001±0.002	0.023±0.010	0.001±0.002	0.005±0.004	0.013±0.008	0.002±0.003
	4	0.001±0.002	0.008±0.006	0.004±0.005	0.001±0.002	0.0±0.0	0.002±0.004	0.005±0.004	0.001±0.002	0.001±0.002	0.009±0.006
	5	0.008±0.007	0.001±0.002	0.003±0.004	0.011±0.011	0.003±0.004	0.0±0.0	0.009±0.007	0.001±0.002	0.002±0.004	0.005±0.006
	6	0.008±0.008	0.003±0.005	0.003±0.005	0.001±0.002	0.005±0.006	0.004±0.006	0.0±0.0	0.0±0.0	0.004±0.004	0.0±0.001
	7	0.0±0.001	0.0±0.001	0.004±0.005	0.002±0.003	0.005±0.006	0.001±0.003	0.0±0.0	0.0±0.0	0.002±0.004	0.008±0.009
	8	0.010±0.008	0.006±0.005	0.007±0.008	0.011±0.009	0.004±0.004	0.010±0.009	0.001±0.003	0.004±0.005	0.0±0.0	0.002±0.005
	9	0.002±0.003	0.001±0.002	0.002±0.004	0.002±0.003	0.009±0.007	0.002±0.005	0.0±0.001	0.011±0.008	0.002±0.004	0.0±0.0

In order to more easily visualize and interpret the confusion matrix, it can be represented as a heat map corresponding to higher values.

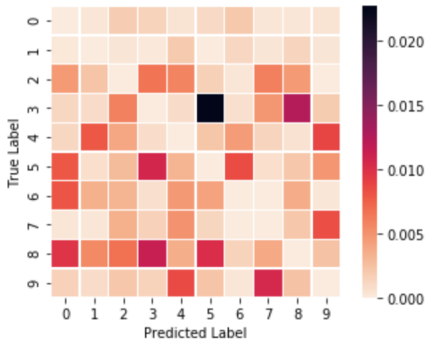


Figure 1: Confusion Matrix Visualization

In [Fig. 1], it can be observed that the highest proportion of mistakes in the particular iteration of 20 runs occurs with respect to 3 being misclassified as 5. Some other common mistakes include 3 confused with 8, 8 with 3, 5 with 3, 8 with 0, 8 with 5, and 9 with 7. These observations are sensible as the curvature of such digits could be construed to look similar to each other. For example, 8, 3, and 5 appear to be most commonly confused with one another, likely due to their similar shapes. Another possible explanation for proportions of mistakes could be the number of data points for each class. If some classes have a significantly smaller number of data points in the train set, they may be more likely to be misclassified, as the perceptron may not learn those classes as well.

4. In order to find five images that are hardest to predict, for each run, the model is trained on the train set and then tested on the entire dataset and the number of misclassifications for each image is recorded. Over 20 runs, the images yielding the highest number of misclassifications during testing are determined to be the hardest images to predict.

Figure 2: Images with highest prediction error

The images most commonly misclassified are sensible. The first and fifth image would likely be misclassified by humans as well, as the images appear to resemble the digits 1 and 0 rather than their true labels 4 and 5. The third image's true label is 6, but is likely more difficult to predict due to its rotation (perhaps being mistaken for 9). The second and fourth images, with true labels 8 and 3, may also be easily confused with 0 and 5.

5. Another common kernel function is that of the Gaussian kernel, defined as $K(\boldsymbol{p}, \boldsymbol{q}) = e^{-c\|\boldsymbol{p}-\boldsymbol{q}\|^2}$, where the scale factor c represents $\frac{1}{2\sigma^2}$ (σ^2 being the variance). The Gaussian kernel is powerful in the sense that it allows for mapping the input to an infinite-dimensional feature space. As $c \to \infty$, the resulting matrix becomes the identity matrix, and as $c \to 0$, the resulting matrix has all elements equal to 1. In either of these cases, any relevant information about the data is lost. Thus, an intermediate value is crucial for having a appropriate kernel matrix. One common strategy for determining c is the use of a median heuristic, an empirical value based on the data [3]. Specifically, the median of X is computed such that $c = \frac{1}{\text{med}(X)}$. Using this method for the given dataset, an initial value for c is set to 0.064. Through experimental observations around this starting point, a range of values are determined for the set of parameters: $S = \{0.004, 0.014, 0.024, 0.034, 0.044, 0.054, 0.064\}$.

Basic results are found by performing 20 runs for each kernel parameter c in S. For each run, a random train-test split of 4:1 is performed, the perceptron is trained on the training set for 15 epochs,

and the test error is computed for the train and test sets. The mean and standard deviation of the train and test error are then computed over the 20 runs for each parameter. From initial experimental observations, the maximum number of epochs to converge over all parameters is found to be 15. Thus, training is performed over 15 epochs to ensure convergence at each value of c. Notably, the model converges faster when utilizing the Gaussian kernel as opposed to the polynomial kernel.

Table 4: Basic results for Gaussian kernel scale factor.

c	Train Error	Test Error
0.004	0.00095±0.00065	0.03116±0.00478
0.014	0.00011±0.00009	0.02382±0.00403
0.024	0.00011±0.00006	0.02599±0.00387
0.034	0.00007±0.00008	0.02987±0.00323
0.044	0.00005±0.00008	0.03441±0.00369
0.054	0.00001±0.00004	0.03919±0.00448
0.064	0.00000±0.00000	0.04296±0.00397

The initial results display a trend of decreasing train and test error up to $c = 0.014$ and then a trend of decreasing train error and increasing test error afterwards. This suggests that $c = 0.014$ may be optimal as higher values of scale factors c may overfit the data.

For 20 runs with different random 4:1 train test splits, cross-validation is employed to select the optimal parameter c^* to train and test on with the full dataset. c^* is then used to train on the full train set and test on the test set.

Table 5: OVA Gaussian kernel perceptron cross-validation results.

Mean c*	Mean Test Error
0.01702±0.00640	0.02624±0.00309

Notably, the model utilizing the Gaussian kernel appears to perform slightly better than when utilizing the polynomial kernel, as the mean test error reaches 0.02624, as opposed to 0.02769. This may be due to the fact that the Gaussian kernel maps to a infinite dimensional space, as opposed to the polynomial kernel mapping to only a e^d dimensional space [2]. Thus the Gaussian kernel may be more optimal for this data set than the polynomial kernel as it may perform better with data that is not linearly separable.

6. Another method for generalizing the binary perceptron algorithm to multiclass classification is that of One-versus-One (OVO) classification [6]. Given a dataset for K classes, OVO creates $\binom{K}{2}$ binary train sets and classifiers. As suggested by the name, each classifier predicts between one class k versus one other class j. The train set for each k vs. j classifier consists only of the training examples with labels k or j, such that $y = k \rightarrow y_{kj} = 1$, $y = j \rightarrow y_{kj} = -1$. In testing, the predictions of all $\binom{K}{2}$ classifiers are summed by class: for class k, the predictions of each classifier corresponding to k vs. j is added and the predictions of each classifier corresponding to i vs. k is subtracted, resulting in K total 'votes' for each data point. The class k with the highest number of votes is then selected as the prediction at that data point. The update for each classifier is done in the same manner as OVA classification, only with respect to each of the $\binom{K}{2}$ classifiers rather than K classifiers. OVO is commonly used in kernel-based algorithms because kernel functions do not scale in proportion to the size of the training set [2].

Basic results are found by performing 20 runs for each parameter $d = 1, \ldots, 7$, the specified degree of

the polynomial kernel. For each run, a random train-test split of 4:1 is performed, the perceptron is trained on the training set for 30 epochs, and the test error is computed for the train and test sets. The mean and standard deviation of the train and test error are then computed over the 20 runs for each parameter. From initial experimental observations, the maximum number of epochs to converge over all parameters is found to be 30. Thus, training is performed over 30 epochs to ensure convergence at each value of d, given that the kernel perceptron converges at different rates depending on its parameterization. Notably, OVO classification requires more epochs to reach convergence as opposed to OVA classification, perhaps due to the reduced dataset size for each classifier.

Table 6: Basic results for OVO polynomial kernel degree.

d	Train Error	Test Error
1	0.01490±0.00494	0.06207±0.00626
2	0.00026±0.00030	0.03465±0.00486
3	0.00028±0.00031	0.03191±0.00393
4	0.00049±0.00094	0.03285±0.00478
5	0.00013±0.00017	0.03156±0.00354
6	0.00014±0.00021	0.03460±0.00354
7	0.00021±0.00035	0.03325±0.00267

The initial results display a trend of decreasing train and test error up to about $d = 5$ and then a tend of increasing test error afterwards, suggesting overfitting with higher polynomial degrees.

For 20 runs with different random 4:1 train test splits, cross-validation is employed to select the optimal parameter d^* to train and test on with the full dataset. d^* is then used to train on the full train set and test on the test set.

Table 7: OVO polynomial kernel perceptron cross-validation results.

Mean d*	Mean Test Error
3.65000±1.23592	0.03363±0.00542

Notably, the polynomial kernel model utilizing OVO classification appears to perform slightly worse than when using OVA classification with a mean test error of 0.03363 as opposed to 0.02769. This is not unusual as OVO and OVA may perform differently, depending on the dataset, as demonstrated in [Reference: in defense of OVA]. Polynomial OVO also performs worst than Gaussian OVA, which yields a mean test error of 0.02624. Additionally, the mean d^* for OVO (3.65) is slightly lower than that OVA (3.9), but still a similar value. This suggests that different kernel parameterizations could be optimal for varying methods of multiclass classification.

7. Artificial neural networks are important algorithms that have significantly impacted the field of machine learning. Neural networks utilize hidden layers consisting of 'neurons' connected to one another; each neuron receives a weighted sum of outputs from the previous layer and performs its own computation. To update, the gradient of the loss function is calculated by backpropagating through the network. Unlike the original perceptron algorithm, neural networks generalize to multiclass classification. Because the network outputs K probabilities (1 for each class), each y is transformed into a one-hot vector of length K, where for $y = k$, the index k of the vector is 1 and the rest of the vector is comprised of 0's. Additionally, the training input X is modified with an additional dimension of value 1 to act as a bias term. In this experiment, a 2-layer neural network is implemented [4],[2]. In particular, the forward pass is computed for input X, weights W_1, W_2, activation functions sigmoid $S(\boldsymbol{x})$, softmax $\sigma(\boldsymbol{x})$, and layers l_1, l_2 such that

$$S(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{x}}}$$

$$\sigma(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_j^N e^{x_j}}$$

$$l_1 = W_1 X^{\mathrm{T}}$$

$$l_2 = W_2 S(l_1)$$

$$\hat{y} = \sigma(l_2)$$

After one forward pass of the entire data (batch size = N), the gradient is computed by using the chain rule to find the derivative of the forward functions. For each weight matrix, this yields,

$$dW_2 = \frac{1}{K}(\hat{y} - y)S(l_1)^{\mathrm{T}}$$

$$dW_1 = \frac{1}{K}W_2^{\mathrm{T}}(\hat{y} - y)dS(l_1)X$$

$$dS(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{x}}\left(1-\frac{1}{1+e^{-\boldsymbol{x}}}\right)}$$

After computing the gradient, for learning-rate r, the weights are updated according to

$$W_2 = W_2 - r(dW_2)$$

$$W_1 = W_1 - r(dW_1)$$

The 2-layer neural network is parameterized by the number of neurons in the hidden layer (hidden size) and the learning rate. For this implementation, the learning rate is held constant and not cross-validated over because of the differing number of epochs required for each learning rate to converge. From initial experimentation, a learning rate of 1 is selected as optimal, from the perspectives of computation time and accuracy. The range of hidden size was also determined by initial experimentation suggesting a larger hidden size would yield higher accuracy.

Initial results are found by performing 20 runs for each parameter $h = \{120, 140, 160, 180, 200, 220, 240\}$, the hidden size. For each run, a random train-test split of 4:1 is performed, the neural network is trained on the training set for 2000 epochs, and the test error is computed for the train and test sets. The mean and standard deviation of the train and test error are then computed over the 20 runs for each parameter. From initial experimental observations, the maximum number of epochs to converge is found to be 2000. Thus, training is performed over 2000 epochs to ensure convergence.

Table 8: Basic results for neural network hidden size.

Hidden Size	Train Error	Test Error
120	0.00220±0.00048	0.08430±0.00468
140	0.00127±0.00046	0.08446±0.00565
160	0.00067±0.00027	0.08341±0.00581
180	0.00050±0.00022	0.08505±0.00540
200	0.00040±0.00019	0.08750±0.00532
220	0.00030±0.00016	0.08849±0.00576
240	0.00022±0.00011	0.08704±0.00620

The initial results display a trend of decreasing train error and test error with increasing hidden size up to $h = 160$ and then a trend increasing test error with decreasing train error, suggesting overfitting with larger hidden sizes.

For 20 runs with different random 4:1 train test splits, cross-validation is employed to select the optimal parameter h^* to train and test on with the full dataset. h^* is then used to train on the full train set and test on the test set.

Table 9: Neural network cross-validation results.

Mean Hidden Size	Mean Test Error
235±10.72381	0.09065±0.00618

The neural network yields a mean test error of 0.09065, performing significantly worse than any of the multiclass perceptrons. This may be due to a neural network's capacity to overfit the data, especially as the number of neurons in the hidden layer increases. The data itself may be approximately linearly separable, in which case using a neural network would be unlikely to improve performance. Overfitting of the neural network can be observed in [Table 8], as train error is significantly smaller than test error and train error continues to decrease as test error increases with increasing hidden size.

A support vector machine (SVM) is another type of binary classification algorithm apart from the perceptron. It works by finding a linear separator that maximizes the distance between the closest data points (which make up the support vector). The following implementation utilizes a soft-margin SVM, allowing some misclassifications, which is more suitable if the data is not linearly separable. In order to generalize the SVM to K classes, OVO is again employed in the same manner as explained previously.

In the linearly nonseparable case, the goal of the SVM is to

$$\text{Minimize } \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} + C\sum_{i=1}^{m}\xi_i$$

$$\text{subject to } y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b) \leq 1 - \xi_i,$$

$$\xi_i \leq 0,\ i = 1, \ldots, m$$

This can be solved by determining the saddle point of the Lagrangian function. When substituting, it yields a final problem of

$$\text{Maximize } Q(\boldsymbol{\alpha}) := -\frac{1}{2}\boldsymbol{a}^{\mathrm{T}}A\boldsymbol{\alpha} + \sum_i \alpha_i$$

$$\text{subject to } \sum_i y_i\alpha_i = 0$$

$$0 \leq a_i \leq C,\ i = 1, \ldots, m$$

where $A = \boldsymbol{y}^{\mathrm{T}}K(\boldsymbol{x}, \boldsymbol{x})\boldsymbol{y}$.
The SVM algorithm can be solved using quadratic programming (QP). QP solves functions of the form,

$$\text{Minimize } \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}P\boldsymbol{x} + \boldsymbol{q}^{\mathrm{T}}\boldsymbol{x}$$

$$\text{subject to } G\boldsymbol{x} \leq \boldsymbol{h}$$

$$A\boldsymbol{x} = b$$

Thus, the SVM problem can be rewritten in the above terms and solved using an optimization library, in this case CVXOPT [1]. Specifically, the maximization of $Q(\boldsymbol{\alpha})$ is multiplied by -1 to become a minimization problem, $\boldsymbol{\alpha}$ is represented by $\boldsymbol{x}$, $\boldsymbol{q}$ becomes a negative coefficient to $\boldsymbol{\alpha}$, the bounds $0 \leq \alpha_i \leq C$ are represented by G and $\boldsymbol{h}$, and $\boldsymbol{y}$ is represented by A. Transforming the SVM problem to QP terms yields

$$P = \begin{bmatrix} y_1 y_1 K(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & y_m y_1 K(\boldsymbol{x}_m, \boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ y_1 y_m K(\boldsymbol{x}_1, \boldsymbol{x}_m) & \dots & y_m y_m K(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix}, \ P \in R^{m \times m}$$

$$\boldsymbol{q} = \begin{bmatrix} -1 \\ \vdots \\ -1 \end{bmatrix}, \ \boldsymbol{q} \in \mathbb{R}^m$$

$$G = \begin{bmatrix} -1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & -1 \\ 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{bmatrix}, \ G \in R^{2m \times m}$$

$$\boldsymbol{h} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ C \\ \vdots \\ C \end{bmatrix}, \ \boldsymbol{h} \in R^{2m}$$

$$A = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \ A \in R^m$$

$$b = [0]$$

Solving the following problem for $\boldsymbol{\alpha}$ gives us values that can then be used to determine the support vectors. In this particular implementation, a lower threshold (ϵ) of 0.0001 is used and the corresponding points of $\boldsymbol{\alpha}$ above the boundary are used. In testing, the elements of the support vector found are used to compute the prediction,

$$\bar{\boldsymbol{w}} = \sum_{i=1}^{m} \bar{\alpha}_i y_i \boldsymbol{x}_i$$

where $\boldsymbol{x}$ is represented by the kernel of the support vector $\boldsymbol{x}$ with test $\boldsymbol{x}$.

The soft-margin OVO Gaussian kernel SVM is parameterized by the cost C and the kernel scale

factor c. From initial experimentation indicating a larger C would yield more accurate results, a range of parameters is selected $S_C = \{1, 10, 100\}$. Additionally, a subset of the previous Gaussian kernel parameters c is selected $S_c = \{0.009, 0.014, 0.019\}$. Additionally, an ϵ of 0.0001 is selected for the threshold of support vectors from all of the Lagrangians. ϵ is not cross-validated over, primarily due to the additional computational time constraints, and the initial observations that $\epsilon = 0.0001$ appears to perform well.

Initial results are found by performing 20 runs for each combination of parameters in S_C and S_c. For each run, a random train-test split of 4:1 is performed, the SVM is trained using QP [1], and the test error is computed for the train and test sets. The mean and standard deviation of the train and test error are then computed over the 20 runs for each parameter combination.

Table 10: Basic results for soft-margin SVM parameters.

C	c	Train Error	Test Error
1	0.009	0.00477±0.00038	0.02403±0.00257
1	0.014	0.00212±0.00028	0.02304±0.00382
1	0.019	0.00138±0.00020	0.02710±0.00304
10	0.009	0.00017±0.00010	0.02242±0.00318
10	0.014	0.00022±0.00009	0.02355±0.00299
10	0.019	0.00025±0.00005	0.02460±0.00293
100	0.009	0.00011±0.00005	0.02169±0.00292
100	0.014	0.00012±0.00004	0.02250±0.00223
100	0.019	0.00009±0.00006	0.02430±0.00304

The initial results indicate that test error decreases as C increases and test error tends to increase as c increases.

For 20 runs with different random 4:1 train-test splits, cross-validation is employed to select the optimal parameter pair C^*, c^*. C^*, c^* is then used to train on the full train set and test on the test set.

Table 11: SVM cross-validation results.

C*	c*	Mean Test Error
95.5±19.615	0.00902±0.0	0.02183±0.00346

Notably, the SVM performs the best out of all of the algorithms implemented for K classification, yielding a mean test error of 0.02183. Additionally, a higher cost appears to be more optimal as it accepts less mistakes. A scale factor of 0.00902 also performs best for the corresponding cost function. An SVM is powerful in high-dimensional spaces, as it seeks to find an optimal separating hyperplane within the data. However, training an SVM can be very computationally expensive, as QP has a high time complexity; testing, on the other hand, is efficient.

In summary, the OVO soft-margin SVM has the lowest mean test error of 0.02183 out of the three algorithms (kernel perceptron, 2-layer neural network, SVM), but has a tradeoff in its high computational complexity $O(N^4K(K-1)/2)$ (where N is number of data points and K is the number of classes). Alternatively, the kernel perceptrons yield mean test errors of 0.02769 (polynomial OVA), 0.02624 (Gaussian OVA), and 0.03363 (polynomial OVO), which are not as precise as the SVM, but instead have a significantly smaller computational complexity of $O(EKN)$ for OVA and $O(ENK(K-1)/2)$ for OVO (where E is the number of epochs), given that the data is run on no more than 30 epochs for any one model variant. Finally, the 2-layer neural network yields the lowest mean test error of 0.09065. It also is very inefficient with a computational complexity of $O(EHKN)$ (where H is the

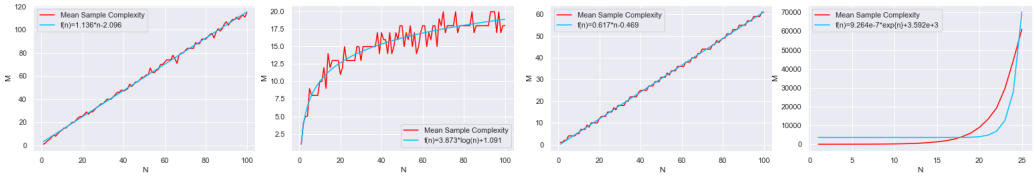
hidden-layer size) as it takes approximately 2000 epochs to converge. Thus, overall the Gaussian OVA kernel perceptron appears to be optimal in balancing accuracy with computation time, while the OVO soft-margin SVM is optimal in terms of accuracy.

2 PART II

2.1 Questions

1. (a) In this particular setting, the data set is an 'expert' problem, where the first dimension of the input $\boldsymbol{x}$ corresponds to the label y, regardless of the dimensionality of $\boldsymbol{x}$. Specifically, $\boldsymbol{x}$ is sampled uniformly at random from $\{-1, 1\}^n$ and each label is defined as $y_i := x_{i,1}$. The sample complexity is defined as being the minimum number of examples (m) required to train on in order to reach a generalization error no greater than 0.1 for data dimension n. The sample complexity can be estimated for various algorithms with experimental observation, through estimating the minimum mean m for each n in which test error is less than or equal to 0.1.

   The perceptron, winnow, least squares, and 1-nearest neighbor algorithms are implemented and used to compute the sample complexity for dimensions of n. Additionally, a best fit line is calculated and plotted to estimate each algorithm's sample complexity function $m \approx f(n)$.



(a) Perceptron (b) Winnow (c) Least Squares (d) 1-Nearest Neighbor

Figure 3: Sample Complexity

   (b) Computing exact sample complexity for the algorithms above would be extremely computationally expensive. Instead, to estimate the complexity, the test error is computed for n, m pairs in order to find the minimum value of m that yields less than or equal to 0.1 mean test error. Specifically, for each algorithm, dimension n is iterated through and a search is employed to select the following m at which to compute the estimated generalization error. At each m determined by the search, the generalization error is estimated. For 100 runs, a train set of dimension n and size m and a test set of dimension n and size 1000 are generated uniformly at random, the model is trained, and the test error is computed; the mean test error over 100 runs is then determined the estimated generalization error for the m, n pair. The search terminates when the minimum m generating an estimated generalization error less than or equal to 0.1 is found and the m is selected as the sample complexity at n. The search utilizes a jump search to determine initial upper and lower bounds for the search space of m: upper bound being the minimum m reached where the estimated generalization error of (m, n) is < 0.1 and the lower bound being the maximum m reached where the estimated generalization error of (m, n) is > 0.1. The jump search then breaks and a binary search is performed within the upper and lower bounds to find m. Utilizing a jump and binary search significantly reduces computation time, minimizing the worst case search time complexity from $O(m)$ to $O(\log(m))$ for each n.

   Estimating the generalization error via average test error procured by random sampling works to decrease bias of a particular train-test set, by creating independently identically-distributed sets for each run. Thus, this method aims to generalize to the space of all possible train-test sets. Assuming that the generalization error is approximately accurate for each m, n pair provides a solution for sample complexity: selecting the minimum m with estimated generalization error less than or equal to 0.1.

   This method has some trade-offs and biases. One such limit is that of estimating the generalization error through only 100 runs rather than, for instance, an infinite amount, or at minimum enough runs (and deterministic generation of train sets) to use every possible permutation of the train

set for (m, n). Rather than ensuring each possible permutation of the train set is used, random generation of train sets over 100 runs aims to estimate the distribution of such a computation. Another bias is that of the test set: in order to decrease computation time, a test set of 1000 points is used rather than a test set containing every possible data point. Again, random generation of the test set over 100 runs aims to estimate the distribution of such computation. This method also assumes that 100 runs and 1000 test points will be of sufficient length to generate a representative distribution of the generalization error.

(c) In [Fig. 3], both the sample complexity and the best fit lines are plotted for each model; thus, the sample complexity can be approximated from the functions above as $m \approx f(n)$. From these experimental observations, the sample complexity of the perceptron algorithm appears to grow linearly as a function of dimension in $\Theta(n)$ as $n \to \infty$. The sample complexity of the least squares algorithm also appears to grow linearly in $\Theta(n)$ as $n \to \infty$. Alternatively, the winnow algorithm seems to be logarithmic, having an average bound of $\Theta(\log n)$ as $n \to \infty$. Finally, 1-nearest neighbor appears to grow exponentially in $\Theta(e^n)$ as $n \to \infty$. The winnow algorithm yields the best performance out of the four models, as the logarithmic sample complexity is significantly less expensive than both linear and exponential bounds. The winnow algorithm is followed by the least squares algorithm and perceptron algorithm in terms of performance: the least squares algorithm has a smaller slope than than the perceptron algorithm, but as $n \to \infty$ they are approximately equal in n. Finally, the 1-nearest neighbor algorithm has the worst performance, as it is exponential and becomes extremely expensive as dimensionality grows. This can be attributed to the curse of dimensionality [6],

(d) The upper bound $\hat{p}_{m,n}$ on the probability that the perceptron will make a mistake on the sth example, where $s \in 1, \ldots, m$ is sampled uniformly at random, after being trained on examples $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{s-1}, y_{s-1})$ can be derived. In particular, the probability of a mistake on the sth example can be expressed as the sum of the product of the probability of a particular s and the probability of making a mistake given the weights $\boldsymbol{w}_s$ at example s.

$$P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0) = \sum_{t=1}^{m} P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0 | s = t)P(s = t)$$

s is sampled uniformly at random from 1 to m, $s \in 1, \ldots, m$. Thus,

$$P(s = t) = \frac{1}{m}$$

$$P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0) = \sum_{t=1}^{m} \frac{1}{m} P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0 | s = t)$$

As proved by Novikoff [5], the perceptron mistake bound is defined

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

Thus,

$$\sum_{t=1}^{m} P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0 | s = t) \leq \left(\frac{R}{\gamma}\right)^2$$

R is defined as

$$R := \max_t \|\boldsymbol{x}_t\|$$
$$= \max_t \sqrt{x_{t_1}^2 + x_{t_2}^2 + \ldots + x_{t_n}^2}$$

Because $X = \{1, -1\}^n$,

$$= \sqrt{1 + 1 + \ldots + 1}$$
$$= \sqrt{n}$$

Thus, R, the maximum distance from the center of the hyperplane is constant $\sqrt{n}$ across all $\boldsymbol{x}$. If there exists a vector $\boldsymbol{v}$ with $\|\boldsymbol{v}\| = 1$, the constant γ is defined as $(\boldsymbol{v} \cdot \boldsymbol{x}_t)y_t\gamma$. Thus,

$$\gamma = \min_t (\boldsymbol{v} \cdot \boldsymbol{x}_t)y_t$$

For a weight vector $\boldsymbol{v} = [1, 0, \ldots, 0]$, where $\|\boldsymbol{v}\| = 1$,

$$= x_{t0}y_t$$

Because for all $\boldsymbol{x}$, $x_0 = y$, and $X, Y \in \{-1, 1\}$,

$$= (x_{t0})^2 = y_t^2$$
$$= 1$$

Thus, the upper bound is

$$P((\boldsymbol{w}_s \cdot \boldsymbol{x}_s)y_s < 0) \leq \frac{1}{m}\left(\frac{R}{\gamma}\right)^2$$
$$= \frac{1}{m}\left(\frac{\sqrt{n}}{1}\right)^2$$
$$= \frac{n}{m}$$

(e) Sample complexity increases exponentially with data dimension n for the 1-nearest neighbor algorithm. A simple function $f(n)$ can be derived for the sample complexity. Utilizing and adapting the formulation of sample complexity in [6] to the current problem,
The error of any given hypothesis of the model can be defined as

$$l(h, (\boldsymbol{x}, y)) = 1_{[h(\boldsymbol{x}) \neq y)]}$$

Because the total loss $L(h_s) = \mathbb{E}[1_{[h_s(\boldsymbol{x}) \neq y]}]$, for all of S,

$$\mathbb{E}[L(h_s)] = \mathbb{E}[1_{[h_s(y \neq y']}]$$
$$\mathbb{E}[P(y \neq y')]$$

where y' is the y-value of the nearest neighbor to point $\boldsymbol{x}$.
Setting $\eta(\boldsymbol{x}) = P(y = 1|\boldsymbol{x})$ and solving for $P(y \neq y')$ for any two points within the domain $\boldsymbol{x}$,$\boldsymbol{x}'$,

$$P(y \neq y') = \eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x}')) + (1 - \eta(\boldsymbol{x}))\eta(\boldsymbol{x}')$$
$$= \eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x}) + \eta(\boldsymbol{x}) - \eta(\boldsymbol{x}') + (1 - \eta(\boldsymbol{x})(\eta(\boldsymbol{x}) - \eta(\boldsymbol{x}) + \eta(\boldsymbol{x}'))$$
$$= 2\eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x})) + (\eta(\boldsymbol{x}) - \eta(\boldsymbol{x}'))(2\eta(\boldsymbol{x}) - 1)$$

Because $0 \leq \eta(\boldsymbol{x}) \leq 1$, $|2\eta(\boldsymbol{x}) - 1| \leq 1$. The conditional probability $\eta(\boldsymbol{x})$ is assumed to be c-Lipschitz for some constant $c > 0$ such that for all $\boldsymbol{x}, \boldsymbol{x}' \in X$,

$$|\eta(\boldsymbol{x}) - \eta(\boldsymbol{x}')| \leq c\|\boldsymbol{x} - \boldsymbol{x}'\|$$

To solve for c, the left-hand side can be rewritten as

$$|\eta(\boldsymbol{x}) - \eta(\boldsymbol{x}')| = |P(y = 1|\boldsymbol{x}) - P(y' = 1|\boldsymbol{x})|$$

Because $x_1 = y$,

$$= |P(x_1 = 1) - P(x_1' = 1)|$$
$$= P(x_1 \neq x_1')$$

Rewriting the right-hand side,

$$\|\boldsymbol{x} - \boldsymbol{x}'\| = \sqrt{\sum_{i=1}^{n}(x_i - x_i')^2}$$

If $x_i = x_i' \Rightarrow (1 - 1)^2 = (-1 - -1)^2 = 0$.
If $x_i \neq x_i' \Rightarrow (1 - -1)^2 = (-1 - 1)^2 = 4$.
Thus,

$$\sqrt{4\sum_{i=1}^{n} P(x_i \neq x_i')}$$

$$\sqrt{4P(x_1 \neq x_1') + 4\sum_{i=2}^{n} P(x_i \neq x_i')}$$

It can be observed that

$$2P(x_1 \neq x_1') \leq \sqrt{4P(x_1 \neq x_1') + 4\sum_{i=2}^{n} P(x_i \neq x_i')}$$

$$P(x_1 \neq x_1') \leq \frac{1}{2}\sqrt{4P(x_1 \neq x_1') + 4\sum_{i=2}^{n} P(x_i \neq x_i')}$$

Thus $c = \frac{1}{2}$,

$$|\eta(\boldsymbol{x}) - \eta(\boldsymbol{x}')| \leq \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{x}'\|$$

$P(y \neq y')$ can be upper-bounded,

$$P(y \neq y') \leq 2\eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x})) + \frac{1}{2}\|\boldsymbol{x} - \boldsymbol{x}'\|$$
$$\mathbb{E}[L(h_s)] \leq \mathbb{E}[2\eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x}))] + \frac{1}{2}\mathbb{E}[\|\boldsymbol{x} - \boldsymbol{x}'\|]$$

For a collection of subsets $C_1, C_2, ..., C_r$ of the domain and the set S of m training points, the expected probability of subsets C intersecting with S is

$$\mathbb{E}\left[\sum_{i:C_i \cap S = \emptyset} P(C_i)\right] = \sum_{i=1}^{r} P(C_i) P(C_i \cap S = \emptyset)$$

The probability of C_i not intersecting with S is equal to the probability of not yielding C_i for every point in S. Thus, for any given i,

$$P(C_i \cap S = \emptyset) = (1 - P(C_i))^m$$

For any x, $(1 - x)^y \leq e^{-xy}$. The probability can be bounded

$$P(C_i \cap S = \emptyset) \leq e^{-P(C_i)m}$$

The expectation then has the upper bound

$$\mathbb{E}\left[\sum_{i:C_i \cap S = \emptyset} P(C_i)\right] \leq \sum_{i=1}^{r} P(C_i) e^{-P(C_i)m} \leq r \max_i P(C_i) e^{-P(C_i)m}$$

To find $\max_i P(C_i)e^{-P(C_i)m}$, written in the terms xe^{-xm}, the derivative with respect to x is taken and set to 0,

$$xe^{-mx}dx$$
$$= -xme^{-mx} + e^{-mx} = 0$$
$$x = \frac{1}{m}$$

The maximum value of the function is then found by substituting x with the maximum value $\frac{1}{m}$,

$$\frac{1}{m}e^{-m\frac{1}{m}}$$
$$= \frac{1}{me}$$

Thus,

$$\mathbb{E}\left[\sum_{i:C_i\cap S=\emptyset} P(C_i)\right] \leq \frac{r}{me}$$

Additionally, the probability of a subset C intersecting with S is bounded by 1

$$\mathbb{E}\left[\sum_{i:C_i\cap S\neq\emptyset} P(C_i)\right] \leq 1$$

The domain space can be partitioned into a grid. For some integer T, the length of each box in the grid is $\epsilon = \frac{1}{T}$. Because the grid must span the interval [-1,1] over n dimensions, the total number of boxes is $r = (\frac{2}{\epsilon})^n = (2T)^n$. The maximum distance between two points lying within the range $[0, \epsilon]$ in one dimension is ϵ. Thus, the distance between two points $\boldsymbol{x}, \boldsymbol{x}'$ within the same square is bounded by,

$$\begin{aligned}\|\boldsymbol{x} - \boldsymbol{x}'\| &\leq \sqrt{\sum_i^n (\epsilon)^2} \\ &= \epsilon\sqrt{n}\end{aligned}$$

Because the distance between (-1,1) is 2, the maximum distance between two points in the entire domain is

$$\begin{aligned}\|\boldsymbol{x} - \boldsymbol{x}'\|_{\max} &= \sqrt{\sum_i^n (2)^2} \\ &= 2\sqrt{n}\end{aligned}$$

Thus, the distance between two points $\boldsymbol{x}, \boldsymbol{x}'$ that do not lie in the same square is bounded by the maximum distance

$$\|\boldsymbol{x} - \boldsymbol{x}'\| \leq 2\sqrt{n}$$

Therefore, to find the expected distance between any given $\boldsymbol{x}$ and its neighbor $\boldsymbol{x}'$,

$$\mathbb{E}[\|\boldsymbol{x} - \boldsymbol{x}'\|] \leq \mathbb{E}\left[P\left(\sum_{i:C_i\cap S=\emptyset} C_i\right) 2\sqrt{n} + P\left(\sum_{i:C_i\cap S\neq\emptyset} C_i\right)\epsilon\sqrt{n}\right]$$

$$\mathbb{E}[\|\boldsymbol{x} - \boldsymbol{x}'\|] \leq \sqrt{n}\left(\frac{2r}{me} + \epsilon\right)$$

Because $r = \left(\frac{2}{\epsilon}\right)^n$,

$$\mathbb{E}[\|\boldsymbol{x} - \boldsymbol{x}'\|] \leq \sqrt{n}\left(\frac{2^{n+1}\epsilon^{-n}}{me} + \epsilon\right)$$

Setting $\epsilon = 2m^{-1/(n+1)}$,

$$\frac{2^{n+1}\epsilon^{-n}}{me} + \epsilon = \frac{2^{n+1}2^{-n}m^{n/(n+1)}}{me} + 2m^{-1/(n+1)}$$

$$2m^{-1/(n+1)}\left(\frac{1}{e} + 1\right) \leq 4m^{-1/(n+1)}$$

Thus, substituting back into the expected error yields a final result,

$$\mathbb{E}[L(h_s)] \leq \mathbb{E}[2\eta(\boldsymbol{x})(1 - \eta(\boldsymbol{x}))] + 2\sqrt{n}m^{-1/(n+1)}$$

Solving for m for some error ε,

$$\begin{aligned}\sqrt{n}m^{-1/(n+1)} &\leq \varepsilon \\ (2\sqrt{n})^{n+1}m^{-1} &\leq \varepsilon^{n+1} \\ m &\geq \left(\frac{2\sqrt{n}}{\varepsilon}\right)^{n+1}\end{aligned}$$

Because sample complexity is defined by yielding a generalization error less than or equal to 0.1,

$$\begin{aligned} m &\geq \left(\frac{2\sqrt{n}}{0.1}\right)^{n+1} \\ &= \left(20\sqrt{n}\right)^{n+1} \end{aligned}$$

3 References

References

[1] M. Andersen and L. Vandenberghe. Cvxopt, https://cvxopt.org/.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] D. Garreau, W. Jitkrittum, and M. Kanagawa. Large sample analysis of the median heuristic. *arXiv: Statistics Theory*, 2017.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016.

[5] A. B. Novikoff. On convergence proofs on perceptrons. Polytechnic Institute of Brooklyn.

[6] S. Ben-David S. Shalev-Shwartz. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.