Supervised Learning (COMP0078) Coursework 2 Report

Yoga Advaith Veturi
zchayav@ucl.ac.uk

January 11, 2021

1 PART 1

The source code for this section is provided in `source_code_p1.ipynb`, which is attached with the submission.

1.1 Kernel Perceptron for Handwritten Digit Classification

Handwritten Digit Classification is a task that is widely regarded as the "hello-world" of pattern recognition and machine learning. The MNIST Dataset is popularly used to perform the same and consists of 60,000 training images and 10,000 test images. In this exercise however, only a small subset, containing 9300 records is used to train various classifiers. A few sample images from the data subset are shown below.

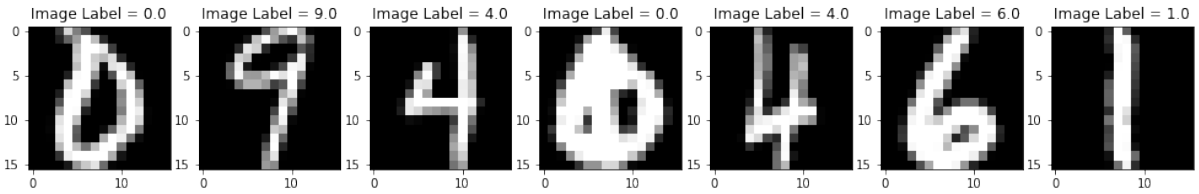

Figure 1: A few sample images from the MNIST data subset. There are totally 9 classes of digits, labelled 0 to 9.

This exercise primarily focuses on a kernel-ized implementation of the perceptron algorithm. A kernel function maps the data to higher dimensions, which allows us to train complex non-linear functions in a computationally efficient way. Below is a basic outline of the kernelized perceptron algorithm, which is used for 2-class classification.

Algorithm 1: Kernel Perceptron (Binary Classification)

Input: $\mathcal{D} = \{(\mathbf{x_1}, \mathrm{y}_1), ..., (\mathrm{x}_m, \mathrm{y}_m)\} \in (\mathrm{R}^n \times \{-1, +1\})^m$

1 Initialize $\boldsymbol{\alpha} = [\mathbf{0}]_{m \times 1}$
2 **for** *t = 1 to m* **do**
3 Recieve datapoint $\mathbf{x_t}$
4 Predict $\hat{y}_t = \text{sign}(\sum_{i=1}^{m} \alpha_i K(\mathbf{x_i}, \mathbf{x_t}))$
5 Recieve true label y_t
6 **if** $y_t.\hat{y}_t \leq 0$ **then**
7 $\alpha_t = y_t$
8 **end**
9 **else**
10 $\alpha_t = 0$
11 **end**
12 **end**

Important variables in algorithm

In the above algorithm, $\boldsymbol{\alpha}$ is the dual solution vector and $K(\mathbf{x_i}, \mathbf{x_t})$ is the kernel function computed between training points $\mathbf{x_i}$ and $\mathbf{x_t}$. Two kernel functions that will be experimented with are the polynomial kernel and the Gaussian kernel. The polynomial kernel is of the form $K(\mathbf{p}, \mathbf{q}) = \langle \mathbf{p}, \mathbf{q} \rangle^d$ where d is the parameter that controls the polynomial degree. The Gaussian kernel is of the form $K(\mathbf{p}, \mathbf{q}) = e^{-c\|\mathbf{p}-\mathbf{q}\|^2}$, where c is the parameter that controls the kernel width.

Model training and Convergence criteria

In terms of model training, the above algorithm processes training examples one at a time in an online fashion for a single pass through the training set. To ensure that the weights are well learned, the model is trained for multiple passes or "epochs". This is done until no further improvement is seen, also known as "convergence". From initial experiments, it was determined that the basic models converged before 20 epochs, hence this was selected as the maximum number of epochs to run the model training function. However, if a model happened to converge much earlier, a convergence criterion was also setup to break out of the training loop. For all experiments, it was decided that if the difference in training error (definition of error is provided in subsequent section) between consecutive epochs was less than an ϵ=0.001 and this was seen over 5 consecutive epochs, then the training for that model is terminated. The ϵ and the number of consecutive epochs is referred to as `epsilon` and `tolerance` in the code.

Evaluating a model

To evaluate a trained model, the final $\boldsymbol{\alpha}$ was used to along with the kernel function to first compute the predictions. This corresponds to line 4 in the algorithm. Note that the kernel function here is computed between the training point $\mathbf{x_i}$ and the test point $\mathbf{x_t}$. The predictions were then compared with the true labels and misclassification error was

computed by

$$Error = \frac{1}{m_{test}} \sum_{t=1}^{m_{test}} \mathbb{I}[\hat{y}_t \neq y_t]]$$

where m_{test} is the number of test examples.

1.2 Generalizing to K classes

The problem in the previous algorithm is that it is a binary classifier. As we are dealing with 10 classes in the MNIST dataset, it is required that the classifier be modified to perform K-class classification. Two strategies to generalize to multiple classes are the One-vs-All and One-vs-One approach. One-vs-all, which is the more common method, is subsequently discussed and One-versus-One is discussed in section 1.2.9.

1.2.1 One-versus-All Classification

With the one-versus-all (OvA) approach, our K-class classification task is divided into K binary classification sub-tasks where in each sub-task, a classifier distinguishes between the k-th class and all the other classes. In other words, K classifiers are trained to predict (0 versus "not 0"), (1 versus "not 1"), up till (K versus "not K"), where in our case, K=10.

To modify the binary classifier into the OvA classifier, the $\boldsymbol{\alpha}$ was now defined to be a $(K \times m)$ matrix where each row is the dual solution for the k-th classifier. The prediction step during training would now be expressed as

$$\hat{y}_{k,t} = \text{sign}(\sum_{i=1}^{m} \alpha_{k,i} K(\mathbf{x_i}, \mathbf{x_t})) \tag{1}$$

where $\alpha_{k,i}$ and $\hat{y}_{k,t}$ correspond to the k-th sub-classifier. As a result, the prediction for instance $\mathbf{x_t}$ during training is a $(K \times 1)$ vector $\hat{\mathbf{y}}_\mathbf{t}$. To compare $\hat{\mathbf{y}}_\mathbf{t} = [\hat{y_{1,t}}, ..., \hat{y_{K,t}}]^T$ with the true label y_t, the label was also encoded as a $(K \times 1)$ vector where the position corresponding to the true class takes +1 and the rest take -1. For example, the encoding for y_t= 2 is $\mathbf{y_t} = [-1, +1, -1, ..., -1]^T$.

The final update step of the algorithm is similar to that of the binary classifier. If some of the elements of vector $\mathbf{y_t}\hat{\mathbf{y}}_\mathbf{t} \leq 0$, then it means that a subset of the K binary classifiers misclassified example t. Therefore, the t-th column in $\boldsymbol{\alpha}$ is updated with $y_{k,t}$ for the rows (classifiers) that misclassified.

When predicting on test points using the trained OvA classifier, the final class is chosen based on relative "confidence" of the classifier predictions. This is done using

$$\hat{y}_k = \arg\max_k \sum_{i=1}^{m} \alpha_{k,i} K(\mathbf{x_i}, \mathbf{x}) \quad \forall k = \{0, ..., \text{K} = 9\} \tag{2}$$

The following section will discuss how the kernel perceptron implementation is made efficient using vectorization.

1.2.2 Vectorizing the kernel perceptron implementation

In terms of computational efficiency, our kernel perceptron implementation has one major issue - the kernel function $K(\mathbf{x_i}, \mathbf{x_t})$ is computed in a serial manner for each data point $\mathbf{x_t}$ when it is received. As the training and test data are fixed for an experiment, it can be seen seen that the kernel function values are also fixed. This suggests that performing multiple epochs of training with this serial approach will result in the same kernel function values being re-computed, which is wasteful.

To avoid repeated computations, we instead pre-compute a kernel matrix $\mathbf{K}$ where $\mathbf{K}_{ij} = K(i, j)$. Required kernel values can then be acquired by just performing indexing operations on $\mathbf{K}$. With the kernel matrix pre-computed, we can now replace the summation terms as seen in equations 1 and 2 with dot products of $\boldsymbol{\alpha}$ and $\mathbf{K}$. Note however that during training, online processing is still performed hence we use the columns of $\mathbf{K}$ and the full $\mathbf{K}$ during evaluation.

The mathematical breakdown of the kernel matrix computation for the polynomial kernel and the Gaussian kernel is subsequently discussed.

Kernel matrix for polynomial kernel
The kernel matrix $\mathbf{K}$ for the polynomial kernel function would have the form:

$$\mathbf{K} = \begin{bmatrix} K(x_1, x_1) & \cdots & K(x_1, x_\ell) \\ \vdots & \ddots & \vdots \\ K(x_m, x_1) & \cdots & K(x_m, x_\ell) \end{bmatrix} = \begin{bmatrix} \langle x_1, x_1 \rangle^d & \cdots & \langle x_1, x_\ell \rangle^d \\ \vdots & \ddots & \vdots \\ \langle x_m, x_1 \rangle^d & \cdots & \langle x_m, x_\ell \rangle^d \end{bmatrix}$$

where m is the number of training points and ℓ is the number of points we compute the kernel function for. Given the training set matrix $\mathbf{X_{train}}$ of shape $(m \times n)$ and a second matrix of points $\mathbf{X_t}$ of shape $(\ell \times n)$, it can be shown that the above expression is equivalent to:

$$\mathbf{K} = (\mathbf{X_{train}} \mathbf{X_t}^T)^d$$

where the matrix raised to power d is an element-wise operation i.e each element in the matrix is raised to d. During training, the $\mathbf{X_t} = \mathbf{X_{train}}$, hence $\mathbf{K}$ will just be an $(m \times m)$ matrix while during evaluation, $\mathbf{X_t} = \mathbf{X_{test}}$, a test matrix of shape $(\ell \times n)$, hence $\mathbf{K}$ is an $(m \times \ell)$ matrix.

Kernel matrix for Gaussian kernel
The Gaussian kernel matrix is represented as follows:

$$\mathbf{K} = \begin{bmatrix} K(x_1, x_1) & \cdots & K(x_1, x_\ell) \\ \vdots & \ddots & \vdots \\ K(x_m, x_1) & \cdots & K(x_m, x_\ell) \end{bmatrix} = \begin{bmatrix} \exp c \parallel x_1 - x_1 \parallel^2 & \cdots & \exp c \parallel x_1 - x_\ell \parallel^2 \\ \vdots & \ddots & \vdots \\ \exp c \parallel x_m - x_1 \parallel^2 & \cdots & \exp c \parallel x_m - x_\ell \parallel^2 \end{bmatrix}$$

Using NumPy, the exponential operation and multiplication with c can be performed element-wise hence we can set these aside. The goal now is to breakdown the $\parallel x_i - x_j \parallel^2$ terms. Using the equation

$$\| x_i - x_j \|^2 = \langle x_i - x_j, x_i - x_j \rangle = \langle x_i, x_i \rangle - 2\langle x_i, x_j \rangle + \langle x_j, x_j \rangle$$

the kernel matrix can be represented as

$$\mathbf{K} = \exp\left[c(\mathbf{B} - 2\mathbf{X_{train}}\mathbf{X_t}^T + \mathbf{C}^T)\right]$$

where

$$\mathbf{B} = \mathbf{diag}(\mathbf{X_{train}}\mathbf{X_{train}}^T)$$
$$\mathbf{C} = \mathbf{diag}(\mathbf{X_t}\mathbf{X_t}^T)$$

The **diag()** operation obtains the diagonals of a matrix. Mathematically, this computation is will have a dimension mismatch, but due to NumPy array broadcasting, this will provide the correct result.

Just like in the polynomial kernel, the $\mathbf{K}$ used in training will be an $(m \times m)$ matrix and $(m \times \ell)$ during evaluation.

Using the ideas developed in the previous two sections, our modified kernel perceptron algorithm for OvA classification is presented below.

Algorithm 2: Kernel Perceptron (Multi-class Classification)

Input: $\mathcal{D} = \{(\mathbf{x_1}, \mathrm{y}_1), ..., (\mathrm{x}_m, \mathrm{y}_m)\} \in \mathbb{R}^n \times \{\text{-1, +1}\}$

Initialize $\boldsymbol{\alpha} = [\mathbf{0}]_{K \times m}$
Precompute kernel matrix $\mathbf{K}$
for *t = 1 to m* **do**
    Recieve datapoint $\mathbf{x_t}$
    Predict $\hat{y_t} = sign(\boldsymbol{\alpha}\mathbf{K}_{1:m,t})$
    Recieve true label y_t and represent as $(k \times 1)$ vector
    for *k = 1 to K* **do**
        if $y_{k,t} * \hat{y_{k,t}} \leq 0$ **then**
            $\alpha_{k,t} = y_{k,t}$
        end
        else
            $\alpha_{k,t} = 0$
        end
    end
end

With this methodology, we now proceed to discuss the results for the experiments conducted on the MNIST dataset.

1.2.3 Experiment 1: Basic Results

The multiclass kernel perceptron algorithm was implemented using a polynomial kernel, where d values in set {1,...,7} were tested. Furthermore, model training and testing with each d parameter was conducted for 20 runs where on each run, a random 80-20 train-test split was generated (in order to smooth the results).

The train set error and test set error averaged over the 20 runs is presented below for each d parameter.

d	Mean Train Error ± STD	Mean Test Error ± STD
1	0.061112 ± 0.011642	0.092446 ± 0.011310
2	0.000780 ± 0.000539	0.032070 ± 0.003640
3	0.000410 ± 0.000302	0.027204 ± 0.004018
4	0.000350 ± 0.000214	0.028898 ± 0.003314
5	0.000773 ± 0.002447	0.027097 ± 0.004159
6	0.000679 ± 0.002190	0.027285 ± 0.003212
7	0.000558 ± 0.001693	0.029059 ± 0.003576

Table 1: Basic results for kernel perceptron averaged over 20 runs for 7 polynomial degrees of polynomial kernel

1.2.4 Experiment 2: Performing 20 runs of 5-fold CV

To identify the best d parameter from our set of d values, 5-fold cross validation was performed on each value. For a random train-test split, the following steps are performed for a single parameter:

1. Shuffle the training set. (To reduce bias)
2. Divide training set into k=5 groups.
3. Select first group as the "validation" set and use the remaining k-1 groups as the training set.
4. Train the kernel perceptron on the training set with the specific kernel parameter.
5. Compute error on the validation set for that fold.
6. Repeat steps (3-6) on all groups.
7. Average the validation set error on all 5 folds to get the mean validation error for that parameter value.

The best kernel parameter d* was chosen to be the one with smallest mean validation set error. Using this best parameter, the model was retrained on the full training set and the error on the test set was recorded. This entire process was performed for 20 runs with different train-test splits, thus 20 d* values and their test errors were produced. This is shown in the Table 2 below.

	d*	Test Error
Run 1	4.0	0.036559
Run 2	5.0	0.026344
Run 3	4.0	0.023118
Run 4	5.0	0.034946
Run 5	5.0	0.029032
Run 6	6.0	0.030108
Run 7	3.0	0.023118
Run 8	5.0	0.027419
Run 9	3.0	0.022581
Run 10	5.0	0.031720
Run 11	4.0	0.024194
Run 12	5.0	0.033871
Run 13	4.0	0.020968
Run 14	6.0	0.027419
Run 15	4.0	0.030645
Run 16	5.0	0.030645
Run 17	3.0	0.030108
Run 18	4.0	0.025806
Run 19	4.0	0.026344
Run 20	5.0	0.027957

Table 2: Results for 5-fold cross validation performed for 20 runs with Kernel Perceptron and polynomial kernel

From these results, we obtain the following:
Mean d* $\pm$ STD $= 4.45 \pm 0.89$
Mean Test Error $\pm$ STD $= 0.028 \pm 0.0043$

1.2.5 Confusion Matrices

For each d* that was determined from each of the 20 runs, the trained model was used to generate predictions on the testing set. Using these predictions, 20 confusion matrices were generated where each is a (10×10) matrix containing a record of the mistakes based on the true and predicted label. These 20 matrices were averaged to produce the final confusion matrix shown in Figure 2.

1.2.6 Finding Hardest Images to Classify

While our trained models have shown the satisfactory performance, as indicated by the numerical results in Table 2, it is essential to understand which images it struggles to classify. This is investigated relative to the cross validation experiments, when the best parameter d* is retrained. We define "difficult" images to be those that regardless of whether they are in the training or test set, are misclassified on every run. Considering this definition, the method to find the hardest images is as follows:

1. Initialize a "mistakes" vector which is the size of the entire dataset i.e. 9298×1

Figure 2: Confusion matrix generated from averaging 20 matrices for all 20 runs. Values are represented as means ± std

2. On every experiment, generate a random train-test split and obtain a "lookup" table containing the indices of the examples that were being used for training and testing.
3. Iterate over d values through cross validation and find best value d*
4. Retrain with d*
5. Compute predictions on training+test set.
6. If the prediction was incorrect, lookup that image's index and add 1 to the corresponding position in the mistakes vector.
7. Repeat for all 20 runs.
8. Identify top 5 images in "mistakes" vector with highest number of mistakes.

Using this method, the top 5 images that hardest to predict are visualized in Figure 3.

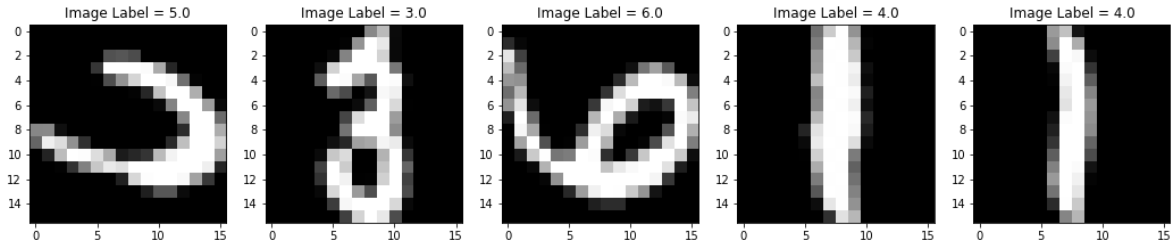

Figure 3: Top 5 images in dataset that are hardest to predict

It is not surprising that these images are hard to predict. The last two images resemble the digit "1" however the true label is "4". The second image is a "3" but very closely

resembles an "8" which could also lead to a misclassification. Finally the "6" and "5" are oddly written. It is possible however that through data cleaning methods and image augmentations, the predictions could be improved.

1.2.7 Repeating Experiments 1 and 2 with Gaussian Kernel

Experiments 1 and 2 are now repeated using the Gaussian kernel. Here, the parameter of interest is is c. From initial experiments, it was seen that models tended to overfit easily for $c \geq 1$. For this reason, a smaller range was chosen : $c = \{2^{-10}, 2^{-9}, 2^{-8}, ..., 2^{-4}\}$.

The basic results for Experiment 1 are presented below.

c	Mean Train Error ± STD	Mean Test Error ± STD
2^{-10}	0.023528 ± 0.007656	0.055215 ± 0.007788
2^{-9}	0.006521 ± 0.004979	0.039516 ± 0.007268
2^{-8}	0.000262 ± 0.000173	0.031022 ± 0.003384
2^{-7}	0.000370 ± 0.000255	0.027500 ± 0.004295
2^{-6}	0.000417 ± 0.000786	0.024677 ± 0.002809
2^{-5}	0.000182 ± 0.000310	0.027796 ± 0.003186
2^{-4}	0.000074 ± 0.000079	0.043414 ± 0.003650

Table 3: Basic results averaged over 20 runs using Kernel perceptron with Gaussian kernel.

5-fold cross validation was then performed to find the best parameter c*. The results per run of the experiment are presented in Table 4.

From these results, we obtain the following:
Mean c* ± STD = 0.014453125 ± 0.002862090222910338
Mean Test Error ± STD = 0.025376344086021508 ± 0.003644744748016605

1.2.8 Comparing the Polynomial and Gaussian kernel results

Comparing the final results of Experiment 2 of the Gaussian kernel with the polynomial kernel, it can be seen that both achieve a similar performance on the test set (2.58% error for polynomial kernel and 2.54% for Gaussian kernel). The results per run show that for majority of the generated test sets, a $c^* = 2^{-6} = 0.015625$ was found to be the best parameter, while in the polynomial kernel, there was a greater spread in the d values. Furthermore, the spread in the test error was lower in the Gaussian kernel compared to the polynomial kernel. Thus, it could be stated that the Gaussian kernel demonstrates a more stable performance on the dataset and thus, would be a better choice of kernel function for the kernel perceptron.

	c*	Test Error
Run 1	0.015625	0.031183
Run 2	0.015625	0.023656
Run 3	0.007812	0.027419
Run 4	0.015625	0.027957
Run 5	0.015625	0.025806
Run 6	0.015625	0.024731
Run 7	0.015625	0.019355
Run 8	0.015625	0.024194
Run 9	0.015625	0.019892
Run 10	0.015625	0.024194
Run 11	0.015625	0.022581
Run 12	0.015625	0.029570
Run 13	0.015625	0.019892
Run 14	0.015625	0.022043
Run 15	0.015625	0.026344
Run 16	0.015625	0.032258
Run 17	0.007812	0.027957
Run 18	0.007812	0.026344
Run 19	0.015625	0.023656
Run 20	0.015625	0.028495

Table 4: Results for 5-fold cross validation performed for 20 runs with Kernel perceptron and Gaussian kernel.

1.2.9 One-versus-One Classification

Another approach to perform multiclass classification is One-vs-One(OvO). Here, we independently train binary classifiers for each pair of classes in our dataset. In other words, we train classifiers in the set {"0 vs 1", "0 vs 2",...,"1 vs 2", "1 vs 3",...,"8 vs 9"}. Totally, there are $\binom{K}{2} = \frac{K(K-1)}{2} = 45$ classifiers.

Model Training

In order to train the models, the first step was to partition the dataset into subsets where each subset contained the class pairs for the specific classifier. For example, to train the"0 vs 1" classifier, the data points labelled "0" and"1" were extracted and used as the training dataset. Then, using the binary classifier Algorithm 2 presented at the very beginning of this report, the individual classifiers were trained. Note that the labels of the data for each classifier were converted to +1 or -1 for model training and a lookup table was used to keep track of the mapping between the original labels and the +1, -1 labels.

Prediction and Evaluation on test points

To test the trained model, a voting based method was implemented. A $(45 \times m_{test})$ "class predictions" matrix was created where m_{test} is the number of test points. Thus a column

in this matrix would store the predictions of each classifier for the given test point. Each model would predict either +1 or -1 and using the lookup table, the binary encodings were mapped back to their original labels. The final predicted class for a test point was then determined by selecting the class label with the most occurrences in the columns of the matrix. It must be noted however that in the event of ties, the smaller class was chosen to be the final prediction.

1.2.10 Repeating Experiments 1 and 2 with OvO approach

Using the OvO strategy, Experiments 1 and 2 were repeated again with the polynomial kernel. The basic results are presented below:

d	Mean Train Error ± STD	Mean Test Error ± STD
1	0.228314 ± 0.028689	0.242634 ± 0.027875
2	0.045745 ± 0.023078	0.075968 ± 0.023769
3	0.037026 ± 0.029208	0.070054 ± 0.029915
4	0.018701 ± 0.023504	0.049973 ± 0.023463
5	0.014917 ± 0.018624	0.047392 ± 0.017616
6	0.005122 ± 0.005909	0.036855 ± 0.006365
7	0.004853 ± 0.005313	0.038172 ± 0.006883

Table 5: Basic results averaged over 20 runs for Kernel perceptron and polynomial kernel and One-vs-One classifcation.

The results from the cross validation experiment are presented in Table 6.

From these results, we obtain the final result:
Mean d* ± STD = 6.3 ± 0.8013147091860318
Mean Test Error ± STD = 0.050510752688172046 ± 0.02148023077158161

	d*	Test Error
Run 1	6.0	0.065054
Run 2	7.0	0.034946
Run 3	7.0	0.036559
Run 4	7.0	0.058065
Run 5	7.0	0.055914
Run 6	7.0	0.045161
Run 7	7.0	0.035484
Run 8	7.0	0.037097
Run 9	5.0	0.048925
Run 10	5.0	0.035484
Run 11	6.0	0.038172
Run 12	5.0	0.045161
Run 13	7.0	0.033333
Run 14	6.0	0.061290
Run 15	5.0	0.122043
Run 16	6.0	0.086022
Run 17	6.0	0.034946
Run 18	7.0	0.038172
Run 19	6.0	0.048387
Run 20	7.0	0.050000

Table 6: Results for 5-fold Cross Validation performed for 20 runs using Kernel perceptron with polynomial kernel and One-vs-One classifcation.

1.3 Comparing performance with two other algorithms

Two additional models were implemented to compare with the kernel perceptron. The chosen models are a support vector machine and the K-Nearest Neighbors algorithm.

1.3.1 Support Vector Machines Implementation [1]

The support vector machine (SVM) algorithm aims to find the optimal separating hyperplane $H_{\mathbf{w},b} = \mathbf{w}^T\mathbf{x} + b$ that maximises the margin between two classes of data. The soft margin SVM has an objective function of the form

$$\min \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i$$

$$\text{s.t} \quad y_i(\mathbf{w}^T\mathbf{x_i} + b) \geq 1 - \xi_i,\ \xi_i \geq 0,\ \forall i = 1, ..., m$$

The parameter C controls the trade-off between minimizing $\mathbf{w}^T\mathbf{w}$ and training error defined by $\sum_{i=1}^{m}\xi_i$. Greater the C would mean that higher weight is given to classifying

[1] The code implementation of the SVM algorithm is provided in `svm_manual_implementation.ipynb`. This code is adapted from the basic implementation provided on Mathieu Blondel's Github repo : https://gist.github.com/mblondel/586753

examples correctly, hence the harder the margin of the SVM. Thus, C is inversely proportional to a regularizing parameter λ as seen in ridge regression.

Computing dual solution
Using the method of Lagrange multipliers, the above optimization problem can be framed as a quadratic programming problem of the form

$$\max Q(\boldsymbol{\alpha}) := -\frac{1}{2}\boldsymbol{\alpha}^T \mathbf{A}\boldsymbol{\alpha} + \sum_i \alpha_i$$

$$\text{s.t} \sum_i y_i \alpha_i = 0, \quad 0 \leq \alpha_i \leq C, \quad \forall i = 1, ..., m$$

where $\boldsymbol{\alpha}$ is a vector containing the Lagrange multipliers and $\mathbf{A} := (y_i y_j \mathbf{x_i}^T \mathbf{x_j} : i, j = 1, ..., m)$.

In order to find the solution $\boldsymbol{\alpha}$, a quadratic programming (QP) solver was used from the `cvxopt` library. This function solves problems of a form:

$$\min(1/2)x^T P x + q^T x$$

$$\text{s.t.} \quad Gx \leq h, \; Ax = b$$

Our maximisation problem was converted into the form of the above minimization equation by negating and then representing the variables in the required form.

Determining support vectors
The support vectors are data points that lie on the margin and thus influence the orientation of the optimal separating hyperplane $H_{\mathbf{w},b}$. These support vectors were determined by acquiring the $(\mathbf{x_i}, y_i)$ for which $\alpha_i > 0$ (in the code implementation, instead of zero, a threshold of 1e-5 was used).

Using the support vectors, the weights and bias could be computed by

$$\mathbf{w} = \sum_{i=1}^{n_{SV}} \alpha_i y_i \mathbf{x}_i$$

$$b = y_i - \mathbf{w}^T x_i$$

where n_{SV} is the number of support vectors.

Kernelizing the SVM
In order to predict the label of a given example $\mathbf{x_t}$, the formula $y_t = \text{sign}(\mathbf{w}^T \mathbf{x_t} + b)$ could be used. However, given the dual representation of $\mathbf{w}$, this could be reformulated as:

$$y_t = \text{sign}(\sum_{i=1}^{n_{SV}} \alpha_i y_i \langle \mathbf{x}_i \mathbf{x_t} \rangle + b)$$

With this reformulated expression, the inner product could be replaced with kernel functions, thereby allowing us to train complex non-linear functions. Thus, for our experiments the $\mathbf{w}$ variable was ignored and the models were trained and evaluated using $\boldsymbol{\alpha}$ and kernels matrices, as in the case of the kernel perceptron algorithm.

Generalizing to k classes Finally, to generalize our SVM model, which performs binary classification, to k-class classifcation, the algorithm was applied in a one-vs-one fashion in the same way as the kernel perceptron (section 1.2.9).

Repeating Experiments 1 and 2 using SVM

For both Experiments 1 and 2, the Gaussian kernel was used with a fixed $c = 2^{-6}$ and different C (Note, this is the regularizing parameter) parameters were tested from the set

$$C = \{10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}\}$$

The choice for c was based on the observation that it demonstrated the best performance for most runs in the Kernel perceptron experiment. It was decided to only cross-validate over C due to long runtime of the SVM OvO implementation.

The results for Experiment 1 and 2 performed with the hand-coded SVM are presented below. To validate the basic results, separate code was also run using `scikit-learn`'s SVM classifier and results (not presented) were found to be similar.

The Basic results from Experiment 1 are presented below.

C	Mean Train Error	Mean Test Error
0.1	0.043352 ± 0.000982	0.059812 ± 0.003910
1.0	0.001513 ± 0.000285	0.025565 ± 0.003942
10.0	0.000235 ± 0.000058	0.022527 ± 0.003480
100.0	0.000094 ± 0.000062	0.024194 ± 0.002343
1000.0	0.000000 ± 0.000000	0.022419 ± 0.002607

Table 7: Basic Results averaged over 20 runs for SVM using Gaussian kernel with $c = 2^{-6}$

The results of cross validation for Experiment 2 are presented in Table 8.

	C*	Test Error
Run 1	100.0	0.023118
Run 2	1000.0	0.021505
Run 3	10.0	0.026344
Run 4	10.0	0.022043
Run 5	100.0	0.020968
Run 6	10.0	0.022581
Run 7	100.0	0.025269
Run 8	100.0	0.023656
Run 9	1000.0	0.023118
Run 10	1000.0	0.024194
Run 11	1000.0	0.020430
Run 12	100.0	0.026344
Run 13	10.0	0.024731
Run 14	10.0	0.019892
Run 15	1000.0	0.024194
Run 16	1000.0	0.023656
Run 17	1000.0	0.022043
Run 18	100.0	0.019892
Run 19	100.0	0.020968
Run 20	100.0	0.024731

Table 8: Results for 5-fold cross validation performed for 20 runs with SVM and Gaussian kernel with $c = 2^{-6}$

The final result is as follows:
Mean d* $\pm$ STD = 392.5 $\pm$ 458.7956201323543
Mean Test Error $\pm$ STD = 0.02298387096774194 $\pm$ 0.001995508584779877

1.3.2 K-Nearest Neighbors

K-Nearest Neighbors is a simpler algorithm where the prediction of a data point is based on the labels of the K nearest points. To implement this, the Euclidean distance is first computed between the test point $\mathbf{x_t}$ and the points in the training set $\mathbf{x_i}$.

$$d(\mathbf{x_i}, \mathbf{x_t}) = \sqrt{\sum_{i=1}^{n} (\mathbf{x_i} - \mathbf{x_t})^2}$$

Then, the true labels of the K points with the smallest distances are obtained. The most popular label from the set of K nearest points is the class that is finally assigned to the test point.

The parameter of interest for KNN is K, the number of neighbours to use for prediction. For this, powers of 3 were tested : $3^0, 3^1, 3^2, ..., 3^4$. The basic results for Experiment 1 are presented below.

K	Mean Train Error ± STD	Mean Test Error ± STD
1	0.000000 ± 0.000000	0.032984 ± 0.003726
3	0.018681 ± 0.000808	0.034892 ± 0.004387
9	0.034996 ± 0.001114	0.042823 ± 0.003423
27	0.058188 ± 0.001222	0.063817 ± 0.005690
81	0.095173 ± 0.001594	0.099731 ± 0.006007

Table 9: Basic results averaged over 20 runs for K-Nearest Neighbors classifcation

The results for Experiment 2 are presented in Table 10.

	K*	Test Error
Run 1	1.0	0.025806
Run 2	1.0	0.026882
Run 3	1.0	0.032258
Run 4	1.0	0.034409
Run 5	1.0	0.028495
Run 6	1.0	0.030645
Run 7	1.0	0.031183
Run 8	1.0	0.027419
Run 9	1.0	0.031720
Run 10	1.0	0.030645
Run 11	1.0	0.033333
Run 12	1.0	0.037634
Run 13	1.0	0.033871
Run 14	1.0	0.027957
Run 15	3.0	0.033333
Run 16	1.0	0.027957
Run 17	1.0	0.035484
Run 18	1.0	0.033871
Run 19	1.0	0.030645
Run 20	1.0	0.038172

Table 10: Results for 5-fold Cross Validation performed for 20 runs with K-Nearest Neighbors

The final result is:
Mean d* ± STD = 1.1 ± 0.44721359549995804
Mean Test Error ± STD = 0.03158602150537635 ± 0.0034924463200914427

The result that K=1 is the best parameter is surprising. It was initially hypothesized that K=1 would overfit however, it is clearly seen that this value demonstrates an excellent performance on the training as well as test set, comparable to the Kernel perceptron and SVM. This is also notwithstanding the fact that our data is multi-dimensional. These

results could be due to the fact that there are very few outliers in the dataset and the distribution of the handwritten digits exist as linearly separable clusters.

1.4 Final Results: Comparing all algorithms

The final results for all algorithms in this exercise are summarized in Table 11.

Algorithm	**Configuration**	**Test set error (mean ± STD)**
Kernel Perceptron	Polynomial kernel ($d = 4.45$), OvA	0.028 ± 0.0043
Kernel Perceptron	Gaussian Kernel ($c = 2^{-6.11279}$), OvA	0.0254 ± 0.0036
Kernel Perceptron	Polynomial Kernel ($d = 6.3$), OvO	0.0505 ± 0.0215
Support Vector Machine	Gaussian Kernel ($c = 2^{-6}$), OvO	0.0230 ± 0.0020
K-Nearest Neighbors	K=1.1	0.0316 ± 0.003492

Table 11: Final performance results for all models

It can be seen that most models produce a test set error in the range of 2-3% with the kernel SVM producing the smallest error of 2.30%.

The 3 algorithms can also be compared in terms of their time complexities for training. Defining n to be the number of features and m to be the number of examples, the time complexities are listed below:

- Kernel perceptron - The computationally expensive operation here is only the kernel matrix computation, which has $\mathcal{O}(nm^2)$. Hence the time complexity of the kernel perceptron is $\mathcal{O}(nm^2)$.
- Kernel SVM - Here, the kernel computation again takes $\mathcal{O}(nm^2)$. For solving the quadratic programming problem to find the dual solution $\boldsymbol{\alpha}$, the computation is an NP-hard problem, however for a positive definite $\mathbf{A}$, this could be solved in $\mathcal{O}(n^3)$[2]. Thus, the time complexity for SVM training is $\mathcal{O}(nm^2 + n^3)$.
- KNN - Here, there is no training step. For predicting directly on a new point, Euclidean distance is measured between each point and the entire "training" data, hence the complexity is $\mathcal{O}(nm)$.

Solely considering these time complexities, KNN would be the best algorithm as it has a linear run time. However since it produced a slightly greater test error of 3.16%, we instead turn to the next best algorithm which is the Kernel perceptron trained using the Gaussian kernel. This demonstrate a 2.5% test set error and has a quadratic run time, smaller than the cubic run time of the SVM.

[2] https://math.stackexchange.com/questions/112124/computational-complexity-of-solving-a-quadratic-program-with-linear-inequality-c

2 Part II

The source code for this section is provided in `source_code_p2.ipynb`.

2.1 Questions

In this section, the sample complexity of the perceptron, winnow, least squares and 1-NN algorithm is investigated as a function of the number of dimensions n. This is the number of examples m required in order to achieve a generalization error $\leq 10\%$.

1. (a) The sample complexity plots for each of the four algorithms are provided below in Figure 4.

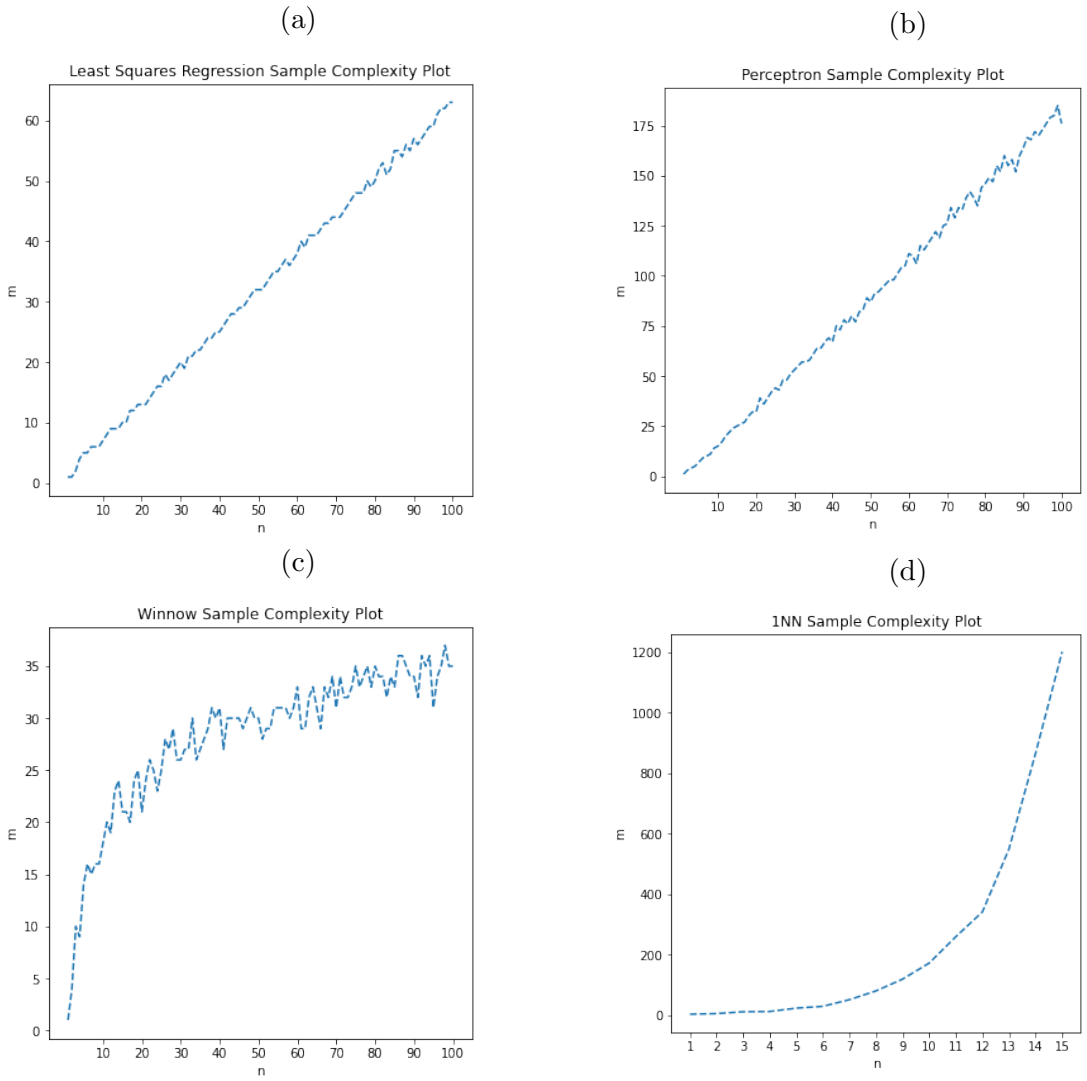

Figure 4: Sample complexity plots for (a) Least squares (b) Perceptron (c) Winnow and (d) 1-Nearest Neighbors. For 1-NN, a smaller n range (1,...,15) was run due to the lengthy runtime. x-axis is the dimensionality of the dataset (n) and y-axis is the minimum number of examples (m) required to achieve a generalization error ≤ 0.10.

(b) In order to determine the sample complexity for an algorithm, the goal of the implementation was to produce a confident estimate by averaging the performance of multiple models trained on multiple datasets. Specifically, for every n value that the sample complexity was computed for, 100 training sets of shape (m_{max}, n) were created where m_{max} is the maximum size that an individual training set was capped at (This was set to 1000, based on initial observations). Next, a variable m was introduced to iterate over the number of examples to train an algorithm on. For example, if m=5, then only rows 1-5 of the 100 training datasets were used for training. m was initialized to 1. During the training procedure, every model was trained on a (m, n) subset of the 100 training datasets using the desired algorithm. This was followed by evaluation, where a single test set of size $(500, n)$ was generated. Each of the 100 trained models was used to generate predictions on the test set. The misclassification error was computed for each model, generating 100 error values. These were averaged to get a final generalization error value. If this generalization error was greater than the threshold of 10%, then m was incremented by 1 and the entire procedure was repeated, but now with $(m + 1)$ rows selected. Else if the generalization error was lesser than 10%, then the sample complexity was set to that value of m and training was next performed on n+1. This procedure is diagrammatically represented in Figure 5.

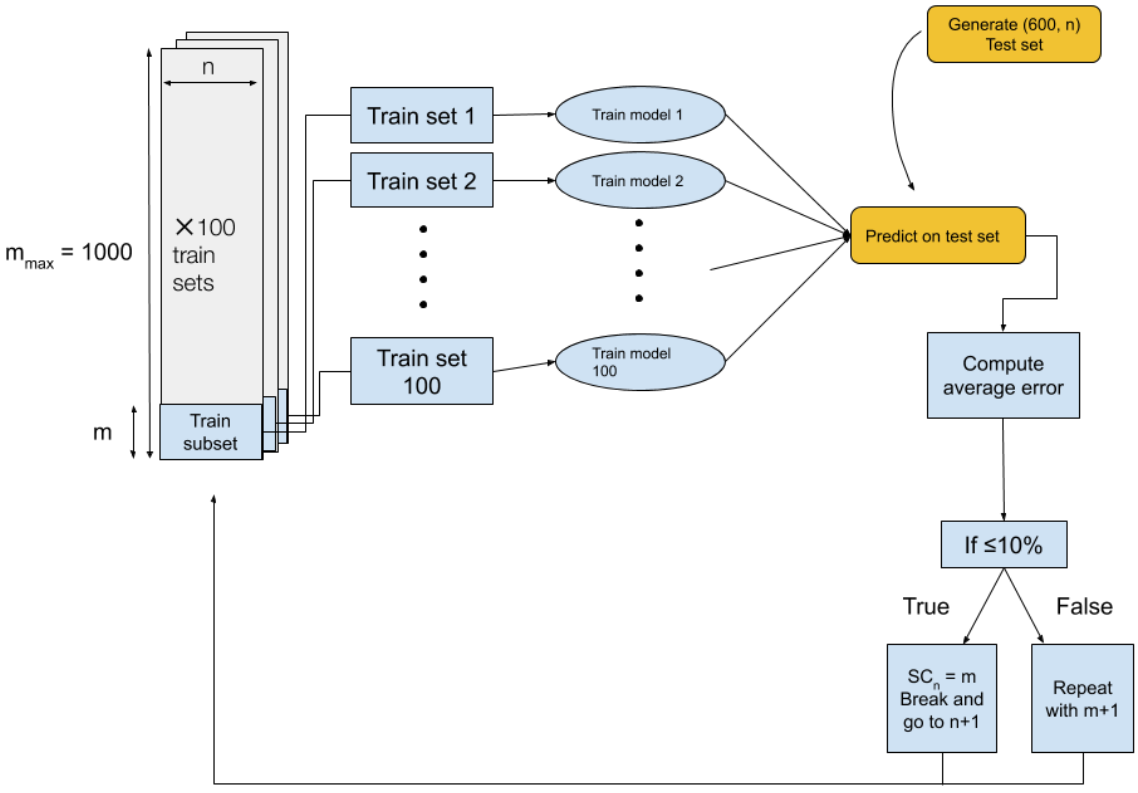

Figure 5: Determining sample complexity for an algorithm

A major trade-off in this method is between the "precision" of the generalization error and the computation time. This is primarily controlled by the number of training sets used and thus, trained models. With fewer trained models, our average test set error will have a higher variability and with that, there is a risk that the generalization error estimate simple arises due to chance. This effect was seen when the experiments were repeated with 30 runs instead of 100 - the sample complexity plots as a function of the dimension n were noisier. However, increasing the number of training sets means that more computational power will be required to train more models, which can be expensive.

It must also be mentioned that a potential bias exists that the generated test sets may contain samples already present within the training set. This violates the assumption that the test set contains unseen data and consequently, the reliability of the generalization error. This could be framed as another trade-off - "Precision" vs "accuracy" of the generalization error - that is controlled by the number of training sets. Increasing the number of training sets means that most samples from the true data distribution are likely to be represented during training, leading to a reduced probability that the test set is unseen and the generalization error is reliable. In other words, the errors from all trained models will be close to each other and "precise" but may not be close to the true value or "accurate".

(c) The plots in Figure 4 can be used to understand the asymptotic behaviour of the sample complexity as $n \to \infty$. From the least squares regression plot, it can be seen that the sample complexity grows linearly with the dimension n, hence it is $\Theta(n)$. For the Perceptron, the sample complexity also grows linearly, hence it is $\Theta(n)$. For the Winnow, the sample complexity grows fast initially however then plateaus as n becomes larger, it is $\Theta(\log n)$. Finally, for the 1-NN algorithm, the number of examples grows exponentially as n increases, hence this can described by $\Theta(2^n)$). Comparing the four algorithms, it can be concluded that the winnow demonstrates the best performance on this classification task as it requires the least examples to achieve a generalization error $\leq 10\%$.

(d) In order to derive a non-trivial upper bound for the probability of error for the perceptron, we first consider that the perceptron has a mistake M bound of

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

where R is the radius of the ball within which the data lies and γ is the size of the margin that separates the data. We now sample an integer s uniformly from $\{1, ..., m\}$ and then select the first $s-1$ examples to train the perceptron with. When the s-th example $(\mathbf{x_s}, y_s)$ is recieved, the probability of misclassification is defined by

$$\hat{p}_{m,n} \leq \frac{R^2}{m\gamma^2}$$

The reason for this is because, letting $B = (\frac{R}{\gamma})^2$, there can be no more than B mistakes, so we can expect that the probability of a misclassification will be no more than B/m on the next example.