COMP0078 Supervised Learning Coursework 2

Douglas Chiang (15055142)
hou.chiang.20@ucl.ac.uk

11^{th} Jan, 2021

1 Part I

1.1 Introduction

Classification is one of the most fundemental task in machine learning. For example, algorithms are used to detect defects when manufacturing semiconductors and predicting protein structures [1]. In this report, we explore and compare the accuracy and efficiency of different algorithms in the realm of supervised learning. In particular, we choose to explore:

1. Kernel perceptron
2. k-nearest neighbors (kNN)
3. Support vector machines (SVM)

We will evaluate the algorithms by using up to 9.300 hand written digits (0 - 9). Since this is a multiclass classification, We generalized the two classes classifying algorithms into k class classifier by exploring two methods for kernel perceptron, one-vs-all and one-vs-one. Two different kernels, polynomial and gaussian kernel will be used for kernel perceptron.In this report, We will first go through the theory of the algorithms above and then compare the results of different approaches mentioned by analyzing hand written digits.

1.2 Theory

In this section, we will first go through the kernel tricks, which help us to map non-linearly separable data to a higher separable dimension. Then we will look into the three algorithms and see how the kernel tricks apply to them.

1.2.1 Kernels tricks

Consider a feature map $\Phi(\mathbf{x})$ which is a function that maps its input to a higher dimension [2]. i.e. $\Phi(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^N$ $(N > n)$ so that it can be used to transforms data from non-linearly separable lower dimension to a linearly separable higher dimension. Kernel K is a function that maps two feature maps $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, defined as the dot product of two feature maps. In many cases, computing the kernels does not require the computation of the feature map explicitly. For example, let $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^n$:

$$\begin{aligned} K(\mathbf{x}_i, \mathbf{x}_j) &= (\mathbf{x}_i \cdot \mathbf{x}_j)^2 \\ &= \left[\sum_{k=1}^{n} x_{ik}x_{jk}\right]\left[\sum_{l=1}^{n} x_{il}x_{jl}\right] \\ &= \sum_{k=1}^{n}\sum_{l=1}^{n} x_{ik}x_{jk}x_{il}x_{jl} \\ &= \sum_{k,l=1}^{n} (x_{ik}x_{il})(x_{jk}x_{jl}) \end{aligned} \tag{1}$$

Then we can see for $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$, $\Phi(\mathbf{x})$ is actually:

$$\Phi(\mathbf{x}) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ \vdots \\ x_1x_n \\ x_2x_1 \\ \vdots \\ x_nx_n \end{bmatrix} \tag{2}$$

And if perform a dot product between the feature vectors, we actually perform the same computation:

$$\begin{aligned} \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle &= \begin{bmatrix} x_{i1}x_{i1} \\ x_{i1}x_{i2} \\ \vdots \\ x_{i1}x_{in} \\ x_{i2}x_{i1} \\ \vdots \\ x_{in}x_{in} \end{bmatrix}^T \begin{bmatrix} x_{j1}x_{j1} \\ x_{j1}x_{j2} \\ \vdots \\ x_{j1}x_{jn} \\ x_{j2}x_{j1} \\ \vdots \\ x_{jn}x_{jn} \end{bmatrix} \\ &= \sum_{k=1}^{n}\sum_{l=1}^{n} x_{ik}x_{jk}x_{il}x_{jl} \\ &= \sum_{k,l=1}^{n} (x_{ik}x_{il})(x_{jk}x_{jl}) \end{aligned} \tag{3}$$

From the above derivation, we see that kernels are functionally equivalent to feature maps.

According to Mercer's Theorem[2][3], for a kernel $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ to be valid, it is necessary and sufficient that for any finite data set $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, the kernel matrix constructed must be positive semi-definite. For a valid kernel matrix $\mathbf{K} \in \mathbb{R}^{m \times m}$:

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \ldots & K(\mathbf{x}_1, \mathbf{x}_m) \\ \vdots & \ddots & \vdots \\ K(\mathbf{x}_m, \mathbf{x}_1) & \ldots & K(\mathbf{x}_m, \mathbf{x}_m) \end{bmatrix} \tag{4}$$

since for every element it is a dot product, so $K(\mathbf{x}_i, \mathbf{x}_j) = K(\mathbf{x}_j, \mathbf{x}_i)$. This means that the kernel matrix is symmetric. Denoting $\phi_a(\mathbf{x})$ as the a^{th} element of $\Phi(\mathbf{x})$, we can see that for any vector $\mathbf{p}$:

$$\begin{aligned} \mathbf{p}^T \mathbf{K} \mathbf{p} &= \sum_i \sum_j p_i K(x_i, x_j) p_j \\ &= \sum_i \sum_j p_i \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) p_j \\ &= \sum_i \sum_j p_i \sum_a \phi_a(\mathbf{x}_i) \phi_a(\mathbf{x}_j) p_j \\ &= \sum_a \sum_i \sum_j p_i \phi_a(\mathbf{x}_i) \phi_a(\mathbf{x}_j) p_j \\ &= \sum_a \left[\sum_i p_i \phi_a(\mathbf{x}_i) \right]^2 \geq 0 \end{aligned} \tag{5}$$

which shows that the kernel matrix is indeed positive semi-definite. This also implies that the kernel need to corresponds to a feature map.

Therefore, kernels opens up opportunities for mult-class classification algorithms such as perceptron which are restricted by the assumption that can only perform well on linearly separable data. By using kernels to map non-linearly separable data from its original space to a space that can separate the transformed data linearly, the accuracy of these linearity restricted algorithms is improved. An figure that can illustrate the above is as below:

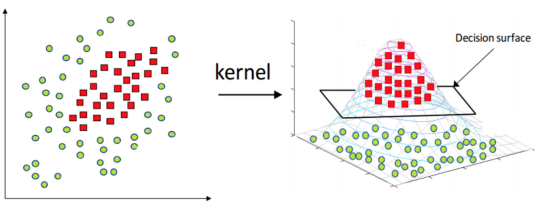


Figure 1: Kernel decision bound [4]

In Figure 1, it is evident that the data points that cannot be separated by a simple plane can be classified after a kernel transformation. In this report polynomial and gaussian kernel are used.

1. *Polynomial kernel* has the form $K_d(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d$. If we expand this by using

binomial theorem, we have:

$$K_d(\mathbf{x}_i, \mathbf{x}_j) = \sum_{i=0}^{d} \binom{d}{i} (\mathbf{x}_i \cdot \mathbf{x}_j)^i$$
$$= \sum_{i=0}^{d} \binom{d}{i} \langle \mathbf{x}_i, \mathbf{x}_j \rangle^i \tag{6}$$

From (5) we can see that using $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ to construct a kernel matrix is positive semi-definite. Therefore, we can see that polynomial kernel in (6) is also a valid kernel because the product of kernels and the sum of kernels also result in kernels.

2. *Gaussian kernel* has the form $K_c(\mathbf{x}_i, \mathbf{x}_j) = e^{-c\|\mathbf{x}_i - \mathbf{x}_j\|^2}$. If we expand the exponent:

$$K_c(\mathbf{x}_i, \mathbf{x}_j) = e^{-c\|\mathbf{x}_i - \mathbf{x}_j\|^2}$$
$$= e^{-c\left[\|\mathbf{x}_i\|^2 + \|\mathbf{x}_j\|^2 - 2\langle \mathbf{x}_i, \mathbf{x}_j \rangle\right]}$$
$$= e^{-c\|\mathbf{x}_i\|^2} e^{-c\|\mathbf{x}_j\|^2} e^{2c\langle \mathbf{x}_i, \mathbf{x}_j \rangle} \tag{7}$$

Since $\|\mathbf{x}_i\|^2$ and $\|\mathbf{x}_j\|^2$ are actually $\langle \mathbf{x}_i, \mathbf{x}_i \rangle$ and $\langle \mathbf{x}_j, \mathbf{x}_j \rangle$, which are also kernels, plus the exponential of a kernel is also a kernel, the gaussian kernel (7) is also a valid kernel.

1.2.2 Algorithms

Perceptron Perceptron is an algorithm for binary classification of data that can be separated by a line (linearly separable), the goal of perceptron is to find a separating hyperplane such that it classify two classes of data [2]. One illustrative figure is as follows:

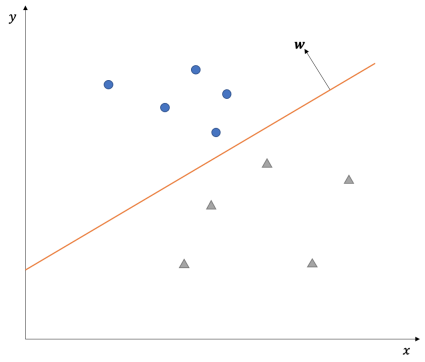


Figure 2: Perceptron illustration

In Figure 2 the orange line separated the data in a space such that on the positive side of the boundary are circles and on the negative side of the boundary are crosses.

Perceptron processes data in an online manner in the sense that it processes one example at a time and we assume that we do not know the size of the data set. When a sample is wrongly classified, the direction of the normal vector $\mathbf{w}$ (weight vector) of the plane will be changed by using the features of the wrongly classified sample. The algorithm that perform such updates is as shown:

Perceptron Algorithm:

Input: A sequence of training samples $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$

1. Initialize weight vector $\mathbf{w}_0 = 0 \in \mathbb{R}^n$
2. For each training sample $(\mathbf{x}_t, y_t)$:
   (a) Predict $\hat{y}_t = sign(\mathbf{w}_t^T \mathbf{x}_t)$
   (b) If true label $y_t \neq \hat{y}_t$, update weight vector $\mathbf{w}_t$: $\mathbf{w}_{t+1} = \mathbf{w}_t + y_t \mathbf{x}_t$
3. Return final weight vector

Since a perceptron classifies two classes ($\{-1, 1\}$), when an error is detected on the a positive sample x_t, the weight vector will be updated as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}_t \tag{8}$$

Then when predicting, the new dot product on this sample will become:

$$\mathbf{w}_{t+1}^T \mathbf{x}_t = (\mathbf{w}_t + \mathbf{x}_t)^T \mathbf{x}_t = \mathbf{w}_t^T \mathbf{x}_t + \mathbf{x}_t^T \mathbf{x}_t > \mathbf{w}_t^T \mathbf{x}_t \tag{9}$$

so we can see that the perceptron will add more weight for a positive sample.
On the other hand, if an error is detected on a negative sample x_t, the weight vector will be updated as:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{x}_t \tag{10}$$

In this case, when predicting, the new dot product on this sample will become:

$$\mathbf{w}_{t+1}^T \mathbf{x}_t = (\mathbf{w}_t - \mathbf{x}_t)^T \mathbf{x}_t = \mathbf{w}_t^T \mathbf{x}_t - \mathbf{x}_t^T \mathbf{x}_t < \mathbf{w}^T \mathbf{x}_t \tag{11}$$

so the perceptron will penalize sample labelled as -1 (negative sample). In other words perceptron is an error driven algorithm. On draw back of this approach is that the classified data must be binary. For this reason, we derive an multiclass perceptron algorithm in Section 1.2.1, we will extend perceptron to kernel perceptron which allows mapping of data to a higher dimensional space.

Now we proceed to fuse the kernel into perceptron algorithm [5], we need to formulate the algorithm into the dual form which express the weight vector $\mathbf{w}$ into a linear combination of the samples $\mathbf{x}_i$. Recall that we use the following formula for prediction upon receiving the t^{th} instance of sample $\mathbf{x}_t$ while training:

$$\hat{y}_t = sign(\mathbf{w}_t^T \mathbf{x}_t) \tag{12}$$

we now expand w_t as a linear combination of the previous $t - 1$ samples:

$$\mathbf{w}_t = \sum_{i=0}^{t-1} \alpha_i \mathbf{x}_i \tag{13}$$

where α_i is the weight of the sample $\mathbf{x}_i$. Now substitute (13) back to (12), we have:

$$\begin{aligned}
\hat{y}_t &= sign(\mathbf{w}_t^T \mathbf{x}_t) \\
&= sign\left(\left[\sum_{i=0}^{t-1} \alpha_i \mathbf{x}_i\right]^T \mathbf{x}_t\right) \\
&= sign\left(\sum_{i=0}^{t-1} \alpha_i \langle \mathbf{x}_i, \mathbf{x}_t \rangle\right) \\
&= sign\left(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)\right)
\end{aligned} \tag{14}$$

with this, the perceptron now becomes kernel perceptron and the algorithm becomes as follow:
Kernel Perceptron Algorithm:

Input: A sequence of training samples $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$

1. Initialize weight vector $\mathbf{w}_0 = 0$ $(\alpha_0 = 0) \in \mathbb{R}^n$
2. For each training sample $(\mathbf{x}_t, y_t)$:
   (a) Predict $\hat{y}_t = sign(\mathbf{w}_t(\mathbf{x}_t)) = sign(\sum_{i=0}^{t-1} \alpha_i K(\cdot, \mathbf{x}_t))$
   (b) If true label $y_t \neq \hat{y}_t$:
      i. Assign $\alpha_t = y_t$
      ii. Update weight vector $\mathbf{w}_t$: $\mathbf{w}_{t+1}(\cdot) = \mathbf{w}_t(\cdot) + \alpha_t K(\cdot, \mathbf{x}_t)$
3. Return final weight vector

Since for the update formula at the t^{th} instance of sample $\mathbf{x}_t$ while training:

$$\begin{aligned}
\mathbf{w}_{t+1}(\mathbf{x}_t) &= \mathbf{w}_t(\mathbf{x}_t) + \alpha_t K(\mathbf{x}_t, \mathbf{x}_t) \\
&= \sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t) + \alpha_t K(\mathbf{x}_t, \mathbf{x}_t) \\
&= \sum_{i=0}^{t} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t) \\
&= \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_t \end{bmatrix}^T \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_t, \mathbf{x}_t) \end{bmatrix}
\end{aligned} \tag{15}$$

Therefore, instead of updating $\mathbf{w}_t$, we can update (assign) $\alpha_t = y_t$ directly if the prediction $\hat{y}_t \neq y_t$ is resulted while training. We can cater this update by creating a vector of size $(1 \times m)$ and then do matrix multiplication with relevant entries in the Gram matrix. We will also expand this vector to $(k \times m)$ where $k = 10$ for the "One vs. All" method and $k = 45$ for the "One vs. One" method. This will be further explain in later sections.

k-Nearest Neighbors (kNN) As mentioned in Section 1.2.2 perceptron is an algorithm that gives a decision boundary that can separate two different classes linearly. There is another kind of algorithm that give predictions by just directly comparing training samples. To explore such algorithms, kNN is chosen in this study.

As the name of the algorithm suggested, kNN is to find $k \in \mathbb{R}^+$ closest training samples to the testing sample [6]. Then the testing sample will be classified as the most frequently appeared class among the k samples. The overall algorithm for kNN is as follows:

kNN Algorithm:
Input: A sequence of training samples $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$ and testing samples $\{\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_t\} \in \mathbb{R}^n$.

1. For every testing sample in $\{\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_t\}$
   (a) Find k closest training samples
   (b) Get the k closest samples' label
   (c) Return the most frequent label as the prediction for the testing sample

When distinguishing between two classes, if we choose k to be an even number, it will potentially lead to undefined prediction because there may be a tie. Using an odd number can solve this issue.

Similar to other ML algorithms, a distance metric must be defined to separate classes. Two common metrics include euclidean distance and cosine similarity. Cosine similarity is chosen in this report as a metric to determine the closest k neighbors of a testing sample. This is because cosine similarity can give a value between $[0, 1]$ where 0 means the two samples are completely different and 1 means the two samples are identical [7]. So we can determine the closest candidates with values between $[0, 1]$ and this can be done irrespective of the size of the feature vector. The cosine similarity between sample $\mathbf{x}_i, \mathbf{x}_j$ is defined as:

$$cos\theta = \frac{\langle \mathbf{x}_i, \mathbf{x}_j \rangle}{\|\mathbf{x}_i\| \, \|\mathbf{x}_j\|} \tag{16}$$

Since kNN is very sensitive to noisy data, when a sample appeared to be very similar to the testing data but actually not the case, cosine similarity will return a value close to 1. This influences the prediction of the algorithm. For example, if there is a tie (For example $k = 7$ and there are three '1's and three '7's), the result of this sample will be critical to break the tie. If unfortunately this sample happens to be a noise (a '1' that is written very similar to '7'), the resultant prediction may be incorrect. This is because kNN does not perform actual training, it is just a direct comparison between testing samples with training samples. For algorithms like perceptron, the training samples are re-feed into the algorithm to fine tune the decision boundary. Therefore, this algorithm is expected to be the worst algorithm among the algorithms that we are going to test.

Support Vector Machine (SVM) SVM is also a linear classifier that used to classify two classes [2]. In contrast to kNN , there can be multiple resultant decision boundaries when classifying two classes when working with perceptron, some may lean more on one class of the

training samples and some may be right at the middle of the two classes of training samples. However, when exposing to unknown data, it may give an incorrect prediction if the decision boundary is very close to one class of data. For this reason, we decided to explore and compare SVM in this study.

In addition to the decision boundary, SVM also takes the margin into consideration, which is defined as the distance between the boundary to the closest data point. An illustration of SVM is in Figure 3 as below:

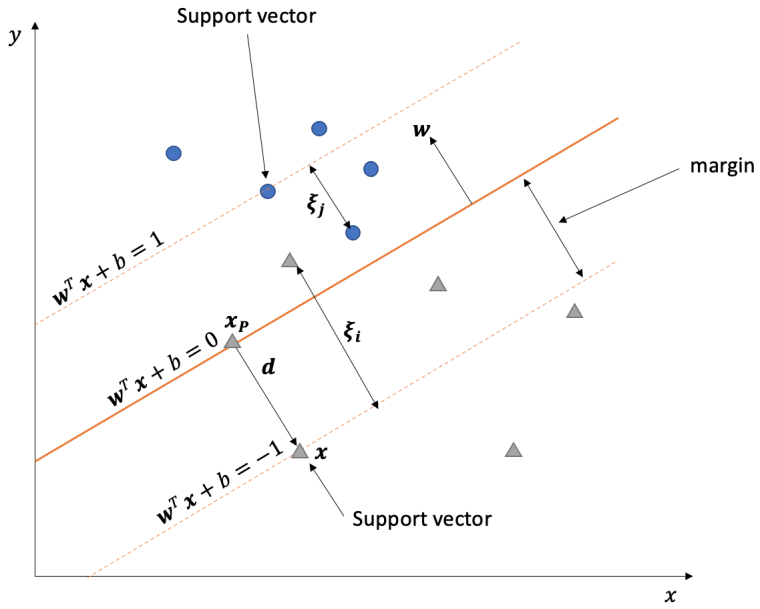


Figure 3: Soft margin SVM

In SVM, the best decision boundaries is the one that gives the widest equal margins between itself and the closest data point of both classes.

We will now first see how we can get the distance of $\mathbf{d}$ between a data point $\mathbf{x}$ and the boundary (hyper-plane) $\mathcal{H}$. For a given hyper-plane $\mathcal{H} = \{\mathbf{x}|\mathbf{w}^T\mathbf{x} + b = 0\}$, we can project the data point $\mathbf{x}$ on the plane in Figure 3 such that the projection $\mathbf{x}_p = \mathbf{x} - \mathbf{d}$ and since the vector $\mathbf{d}$ is parallel to $\mathbf{w}$ we have $\mathbf{x}_p = \mathbf{x} - k\mathbf{w}$ where $k \in \mathbb{R}$.

Furthermore, since $\mathbf{x}_p$ is now a point in the hyper-plane $\mathcal{H}$ we have:

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_p + b &= 0 \\ \mathbf{w}^T(\mathbf{x} - k\mathbf{w}) + b &= 0 \\ k &= \frac{\mathbf{w}^T\mathbf{x} + b}{\mathbf{w}^T\mathbf{w}} \end{aligned} \tag{17}$$

Then with (17) we can have the distance of $\mathbf{d}$:

$$\begin{aligned} \|\mathbf{d}\| &= |k|\sqrt{\mathbf{w}^T\mathbf{w}} \\ &= \left|\frac{\mathbf{w}^T\mathbf{x} + b}{\|\mathbf{w}\|^2}\right| \|\mathbf{w}\| \\ &= \frac{|\mathbf{w}^T\mathbf{x} + b|}{\|\mathbf{w}\|} \end{aligned} \tag{18}$$

With (18) we can define the margin as the distance between the boundary to the closest data point. Suppose we have a set of training data points $\mathcal{S} = \mathbf{x}_1, ..., \mathbf{x}_m \in \mathbb{R}^n$, the margin ρ is then:

$$\rho = \min_{\mathbf{x} \in \mathcal{S}} \frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|} \tag{19}$$

As mentioned earlier in the beginning of Section 1.2.2, the best decision boundaries is the one that has the widest equal margins between itself and the closest data point of both classes. That means we have to maximize the margin ρ subject to a constraint on the hyper-plane $\forall i \ y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 0$:

$$\begin{aligned} \max_{\mathbf{w},b} \rho(\mathbf{w}, b) &= \max_{\mathbf{w},b} \left[\min_{\mathbf{x} \in \mathcal{S}} \frac{|\mathbf{w}^T \mathbf{x} + b|}{\|\mathbf{w}\|} \right] \\ &= \max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|} \left[\min_{\mathbf{x} \in \mathcal{S}} |\mathbf{w}^T \mathbf{x} + b| \right] \end{aligned} \tag{20}$$

Since the hyper-plane is scale invariant, we can make the scale of $\mathbf{w}, b$ such that:

$$\min_{\mathbf{x} \in \mathcal{S}} |\mathbf{w}^T \mathbf{x} + b| = 1 \tag{21}$$

So we have to add this to the hyper-plane constraint:

$$\begin{cases} \forall i \ y_i(\mathbf{w}^T \mathbf{x}_i + b) & \geq 0 \\ \min_{\mathbf{x} \in \mathcal{S}} |\mathbf{w}^T \mathbf{x} + b| & = 1 \end{cases} \rightarrow \forall i \ y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \tag{22}$$

since $\min_{\mathbf{x} \in \mathcal{S}} |\mathbf{w}^T \mathbf{x} + b| = 1$ can be seen as $\min_i y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$, meaning that all the samples are on the correct side of the hyper-plane.
Then, using (21), we now have (20) express as:

$$\begin{aligned} \max_{\mathbf{w},b} \rho(\mathbf{w}, b) &= \max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|} \left[\min_{\mathbf{x} \in \mathcal{S}} |\mathbf{w}^T \mathbf{x} + b| \right] \\ &= \max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|} \\ &= \min_{\mathbf{w},b} \|\mathbf{w}\| \\ &= \min_{\mathbf{w},b} \frac{1}{2} \mathbf{w}^T \mathbf{w} \end{aligned} \tag{23}$$

The last step can be arrived because minimizing $\|\mathbf{w}\|$ is equivalent to minimizing $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ ($\frac{1}{2}$ is introduced for the sake of easier math afterwards)[2]. Concluding the above, to maximize the margin, we solve the following quadratic optimization problem by combining (22) and (23):

$$\begin{aligned} &\min_{\mathbf{w},b} \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ &\text{subject to: } \forall i \ y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \end{aligned} \tag{24}$$

After we tempus the optimal pair of $\mathbf{w}, b$ which maximise the margin ρ, we will find out that there are some training samples that satisfy:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1 \tag{25}$$

Training samples that satisfy (25) are support vectors shown in Figure (3) which determine the shape of the hyper-plane.

Then we can predict a sample $\mathbf{x}_t \in \mathbb{R}^n$ by:

$$\hat{y}_t = sign\left(\mathbf{w}^T\mathbf{x}_t + b\right) \tag{26}$$

So, similar to perceptron, if $\hat{y}_t = +1$, $\mathbf{x}_i$ is classified on the positive side of the hyper-plane and vice versa.

All above works well when the data points of the two classes are clearly separated. However, in most of the cases, some data points can be find in opposite classes' region. So there will be no solution for (24). In this case, we need to allow some elasticity for the SVM by introducing slack variables ξ_i which is essentially the distance between the training data points in the margin and the margin boundary. This allow some training data points to be in the margin or even on the wrong side of the hyper-plane. This is also illustrated as in Figure 3.

With the slack variable ξ_i, the optimization problem in (24) becomes:

$$\begin{aligned} &\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i \\ &\text{subject to: } \forall i \begin{cases} y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{cases} \end{aligned} \tag{27}$$

Here C is the penalty weight for the training data points in the margin or on the wrong side of the hyper-plane. When C is small, the margin is very large so it allows more data to be in the margin. When C is large, the margin is very small and so the SVM now try to get every training data point on the correct side. This is illustrated in Figure 4 bellow:

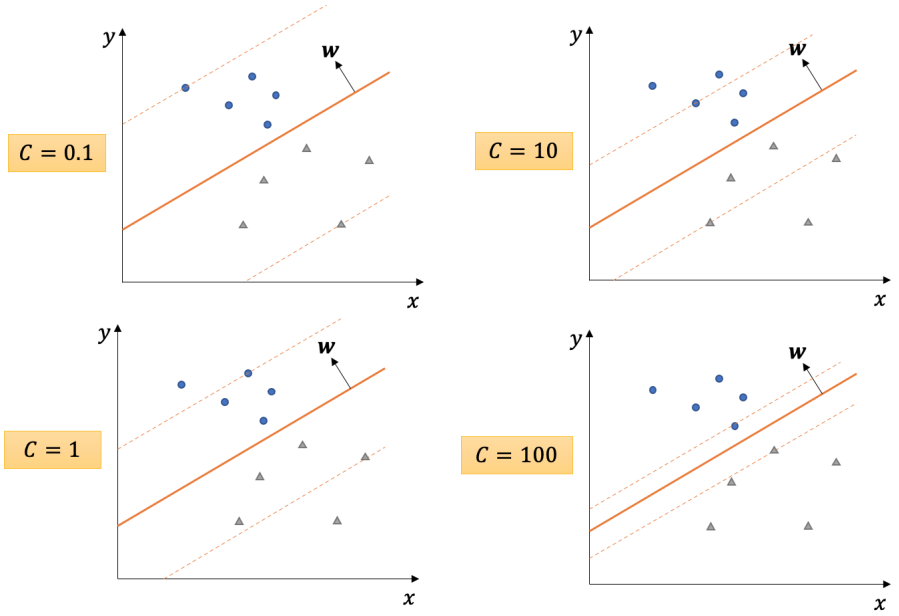


Figure 4: SVM margins with different values of C

Now we extend the problem in (27) to Kernel SVM using dual form. For (27), it is actually solving the saddle point of the Lagrangian function L [2]:

$$L = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) + \xi_i - 1] - \sum_{i=1}^{m}\beta_i\xi_i \tag{28}$$

where $\alpha_i, \beta_i \geq 0$ are Lagragian multipliers.
We then go ahead to minimize L by differentiate with respect to $\mathbf{w}, \xi, \mathbf{b}$:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = 0$$

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i \tag{29}$$

$$\frac{\partial L}{\partial \xi} = C - \alpha_i - \beta_i = 0$$

$$\rightarrow 0 \leq \alpha_i \leq C \tag{30}$$

$$\frac{\partial L}{\partial \mathbf{b}} = -\sum_{i=1}^{m} y_i \alpha_i = 0 \tag{31}$$

It is worth noting that since terms are vanished when $\alpha_i = 0$ in (29), it is a linear combination of support vectors only ($\alpha_i > 0$). Then by substitute (29), (30) and (33) into (28) gives the dual problem, for all training samples $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$:

$$\min_{\alpha_1,...,\alpha_m} \frac{1}{2} \left[\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i \right]^T \left[\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i \right] - \sum_{i=1}^{m} \alpha_i$$

$$\text{subject to: } \forall i \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \end{cases}$$

$$\min_{\alpha_1,...,\alpha_m} \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j - \sum_{i=1}^{m} \alpha_i$$

$$\text{subject to: } \forall i \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \end{cases}$$

$$\min_{\alpha_1,...,\alpha_m} \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{m} \alpha_i$$

$$\text{subject to: } \forall i \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \end{cases} \tag{32}$$

We can see that with the dual problem, a kernel is involved into the calculation. So, after we find the correct set of α_i, we need to find b so that we can proceed with the prediction. From (25), we know that if x_i is a support vector, it satisfies:

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) = 1$$

which means y_i and $\mathbf{w}^T \mathbf{x}_i + b$ are both $+1$ or -1. We now substitute (29) into it, and result

in the following:

$$y_i \left(\left[\sum_{j=1}^{m} \alpha_j y_j \mathbf{x}_j \right]^T \mathbf{x}_i + b \right) = 1$$

$$y_i \left(\sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}_j, \mathbf{x}_i) + b \right) = 1$$

$$\sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}_j, \mathbf{x}_i) + b = y_i$$

$$b = y_i - \sum_{j=1}^{m} \alpha_j y_j K(\mathbf{x}_j, \mathbf{x}_i) \tag{33}$$

Since there are always more than one support vectors, it is worth averaging several values of b using different support vectors. After finding b, we can now express $\mathbf{w}$ as (29) in (26) for the prediction of kernel SVM, and we now have the following when we are predicting a sample $\mathbf{x}_t$:

$$\begin{aligned} \hat{y}_t &= sign \left(\left[\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i \right]^T \mathbf{x}_t + b \right) \\ &= sign \left(\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_t + b \right) \\ &= sign \left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_t) + b \right) \end{aligned} \tag{34}$$

To summarize, the Kernel SVM have the following steps:

Kernel SVM:
Input: A sequence of training samples $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$

1. Solve the dual optimization problem for $\alpha_1, ..., \alpha_m$:

$$\min_{\alpha_1,\ldots,\alpha_m} \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{m} \alpha_i$$

$$\text{subject to: } \forall i \begin{cases} \sum_{i=1}^{m} \alpha_i y_i = 0 \\ 0 \leq \alpha_i \leq C \end{cases}$$

2. Use $\alpha_1, ..., \alpha_m$ and the support vectors to calculate b.
3. Return $\alpha_1, ..., \alpha_m$ and b.
4. Then we can use:

$$\hat{y}_t = sign \left(\sum_{i=1}^{m} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_t) + b \right)$$

   to predict new samples $\mathbf{x}_t$.

In this study, gaussian kernel with fixed exponent constant $c = \frac{1}{n}$ is used to evaluate the performance. This will be further discussed in the methodology part.

1.3 Method

We need to generalize the two classes classifying algorithms to multi-class classifying algorithms since we are going to classify ten different hand-written digits (0 - 9). For this we have to evaluate two methods, "One vs. Rest" and "One vs. One" for this generalization. Furthermore, we optimize the algorithms for large scale training by converting operations to matrix algebra. We evaluate the training performance using large scale datasets (of up to 9.300 samples).

To understand how the algorithms are affected by the hyperparameters, we also experiment on different shape of decision boundaries by changing different parameters in the algorithms and cross validate them to determine the best parameter value that can give the best performance on unseen data.

While simple hit/miss ratio gives an overview of the classification performance, we further investigate the pairwise performance in a multi-class classification. For the perceptron algorithm, a confusion matrix is used to showcase the miss rate of different combinations of predictions and the five hardest to predict digits will also be showcased.

1.3.1 "One vs. Rest"

In the One vs. Rest approach, we will have one classifier per class. In other words, we will have 10 classifiers in total. Each classifier will classify class i as $+1$ and all other classes as -1 [8]. This means that it is very likely to have the number of all other samples out-numbered class i. This may result in imbalanced data sets may result in models that gives a bad predictive performance, especially minority classes. This is because unknown samples can be easily on the other side of the decision boundary.

1.3.2 "One vs. One"

In the One vs. One approach, we build a classifier for every pair of classes. So suppose we have N classes, we will then have $\frac{N(N-1)}{2}$ classifiers [8]. We can see that One vs. One need significantly more classifiers than One vs. Rest. So more computation resources are required for One vs. One. However, since for each classifier we now only have just two classes to compare, One vs. One is less prone to the imbalance data sets problem. In this study, we will have 45 classifiers for the One vs. One method.

1.4 Implementation

The algorithms are implemented using python in a Google colab environment. In this section, the way of optimizing the operations for faster computation will be discussed.

1.4.1 Performance optimization

Given the complexity of operations in this study, if we compute the data one at a time, it will be very time consuming. Therefore, multiple operations are into matrix form in kernel perceptron and kNN.

Kernels Before doing the actual training, as we can see we need to calculate the kernel values for every given data, so it is worth pre-computing the kernel matrix [9] before training the algorithms. Suppose we now have a grand data set $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_m, y_m)\} \in (\mathbb{R}^n \times \{-1, 1\})^m$. We can see that the polynomial and gaussian kernel can be turned into the following matrix operations by putting the data in a vector:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_m^T \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{35}$$

Polynomial Kernel
We can pre-caculate the data matrix using (37).

$$\mathbf{K}_d = (\mathbf{X}\mathbf{X}^T)^d \tag{36}$$

$$= \begin{bmatrix} (\mathbf{x}_1^T\mathbf{x}_1)^d & \dots & (\mathbf{x}_1^T\mathbf{x}_m)^d \\ \vdots & \ddots & \vdots \\ (\mathbf{x}_m^T\mathbf{x}_1)^d & \dots & (\mathbf{x}_m^T\mathbf{x}_m)^d \end{bmatrix} \tag{37}$$

Gaussian Kernel
And for the gaussian kernel, since if we expand it we can get:

$$\begin{aligned} K_c(\mathbf{x}_i, \mathbf{x}_j) &= e^{-c\|\mathbf{x}_i - \mathbf{x}_j\|^2} \\ &= e^{-c[\langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_j, \mathbf{x}_j \rangle - 2\langle \mathbf{x}_i, \mathbf{x}_j \rangle]} \end{aligned} \tag{38}$$

we can compute the gaussian of all the data in two steps:

1. Compute and a vector of all the self dot products:

$$\mathbf{X}_{\langle\cdot\rangle} = [\mathbf{x}_1^T\mathbf{x}_1, \mathbf{x}_2^T\mathbf{x}_2, \dots, \mathbf{x}_m^T\mathbf{x}_m]^T \tag{39}$$

2. Then we can make use of (39) to calculate the gaussian kernel for every data by:

$$\mathbf{K}_c = e^{-c[\mathbf{X}_{\langle\cdot\rangle} + \mathbf{X}_{\langle\cdot\rangle}^T - 2\mathbf{X}\mathbf{X}^T]} \tag{40}$$

$$= \begin{bmatrix} e^{-c\|\mathbf{x}_1 - \mathbf{x}_1\|^2} & \dots & e^{-c\|\mathbf{x}_1 - \mathbf{x}_m\|^2} \\ \vdots & \ddots & \vdots \\ e^{-c\|\mathbf{x}_m - \mathbf{x}_1\|^2} & \dots & e^{-c\|\mathbf{x}_m - \mathbf{x}_m\|^2} \end{bmatrix} \tag{41}$$

We can see that every entries in (37) and (41) are kernels between the i^{th} and j^{th} data. With these two kernel matrices, we can extract relevant entries for training and testing as needed. So we do not need to calculate the kernel repetitively. Therefore, (36) and (40) will be used to compute the kernel matrices for the polynomial kernel and gaussian kernel.

Kernel perceptron In the case of kernel perceptron, we know that for a two class classifier we need to predict using (12) and update using (15) during training. Firstly, we have seen that we can expand the prediction formula for a sample $\mathbf{x}_t$ as:

$$\hat{y}_t = sign\left(\sum_{i=0}^{t-1} \alpha_i K(\mathbf{x}_i, \mathbf{x}_t)\right) \tag{42}$$

So if we put all the α_i in a vector, we can have the following operation:

$$\hat{y}_t = \begin{bmatrix}\alpha_1 & \alpha_2 & \dots & \alpha_{t-1}\end{bmatrix} \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_{t-1}, \mathbf{x}_t) \end{bmatrix} \tag{43}$$

which we can extract relevant entries in the kernel matrix by extracting relevant indices. Now if we have k classes, we can expand the vector containing the α to a $k \times (t-1)$ matrix. This results in the following operation:

$$\hat{\mathbf{y}}_t = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,t-1} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,t-1} \end{bmatrix} \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_{t-1}, \mathbf{x}_t) \end{bmatrix} \tag{44}$$

Since we need to update $\alpha_i = y_i$ when the prediction is wrong and there will be 20 epochs, we can be more aggressive and first initialize a $k \times m$ matrix for the α with all zeros and then extract relevant entries as needed. If we do this, we can actually make $(t-1) \to m$ because all α are zeros beyond the $t - 1^{th}$ entry:

$$\hat{\mathbf{y}}_t = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,m} \end{bmatrix} \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_t) \\ K(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots \\ K(\mathbf{x}_m, \mathbf{x}_t) \end{bmatrix} \tag{45}$$

With (45), we will get a $k \times 1$ vector $\hat{\mathbf{y}}_t$ as the predictions of all k classifiers. We will then convert entries to $+1$ for the resultant values > 0, and -1 otherwise.

For the true label y_t, we will also convert it to a vector $\mathbf{y}_t$ of size $k \times 1$ of $-1, +1$ using one hot, meaning that if the true label is '2', $+1$ is only assigned to the second entry and all others will be -1. Then we can compare $\hat{\mathbf{y}}_t$ and $\mathbf{y}_t$ entry by entry and then $+y_t$ for entries in the matrix containing all the α where values in $\hat{\mathbf{y}}_t$ and $\mathbf{y}_t$ do not agree. So after running all the epochs we will have a matrix of α for us to process the testing data.

There is another similar operation for testing, since when testing we do not require it to be 'online' we can expand the kernel vector to a matrix of size $m \times m_{test}$ where m_{test} is the number of testing samples. With this, the prediction is upgraded to a matrix operation as below:

$$\hat{\mathbf{Y}} = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,m} \end{bmatrix} \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \dots & K(\mathbf{x}_1, \mathbf{x}_{m_{test}}) \\ \vdots & \ddots & \vdots \\ K(\mathbf{x}_m, \mathbf{x}_1) & \dots & K(\mathbf{x}_m, \mathbf{x}_{m_{test}}) \end{bmatrix} \tag{46}$$

This operation in (46) gives a $k \times m_{test}$ matrix containing all the predictions for all testing samples with all k classifiers. After getting these values, we will convert $\hat{\mathbf{Y}}$ to a $m_{test} \times 1$ vector of predicted labels 0 - 9 by locating the largest value in each column. Then we will use this $m_{test} \times 1$ vector to compare with the true labels $\mathbf{Y}$ which is arranged in the same manner directly. We can then count the number of correct predictions and calculate the accuracy and the error rates. In our program we will first evaluate our model using the training data set and then the testing set.

k-nearest neigborhood (kNN) In kNN, we use cosine similarity to find the closest k neighbors. The operation for calculating the cosine similarity is converted to a series of matrix operations. Firstly, similar to the polynomial kernel, we calculate the dot product between all the training samples $\mathbf{X}$ and testing samples $\mathbf{X}_{test}$ by:

$$\mathbf{X}_{\langle\cdot\rangle} = \mathbf{X}_{test}\mathbf{X}^T \tag{47}$$

Second, we will calculate the norm of all the samples in the training set and the testing set:

$$\mathbf{X}_{\|\cdot\|_{train}} = \begin{bmatrix} \sqrt{\mathbf{x}_1^T\mathbf{x}_1} \\ \sqrt{\mathbf{x}_2^T\mathbf{x}_2} \\ \vdots \\ \sqrt{\mathbf{x}_m^T\mathbf{x}_m} \end{bmatrix} \tag{48}$$

$$\mathbf{X}_{\|\cdot\|_{test}} = \begin{bmatrix} \sqrt{\mathbf{x}_1^T\mathbf{x}_1} \\ \sqrt{\mathbf{x}_2^T\mathbf{x}_2} \\ \vdots \\ \sqrt{\mathbf{x}_{m_{test}}^T\mathbf{x}_{m_{test}}} \end{bmatrix} \tag{49}$$

Finally, we make use of (47),(48) and (49) to calculate the similarity across all training samples for all testing samples as a matrix $\mathbf{C}_{sim}$ with size $m_{test} \times m$:

$$\mathbf{C}_{sim} = \frac{\mathbf{X}_{\langle\cdot\rangle}}{\mathbf{X}_{\|\cdot\|_{test}} \otimes \mathbf{X}_{\|\cdot\|_{train}}} \tag{50}$$

After calculating the cosine similarity matrix $\mathbf{C}_{sim}$ we can then find the k largest values for each testing sample as their neighbors and recover their labels in the training labels. The prediction for each testing sample is then the label most frequently appeared in their k neighbors. Since kNN directly compare the testing sample with all the samples in the training set, there is no "One vs. Rest" or "One vs. One" classifying concepts for kNN.

SVM As mentioned in the Section 1.2.2, a gaussian kernel is implemented for SVM. In the gaussian kernel $K_c(\mathbf{x}_i, \mathbf{x}_j) = e^{-c\|\mathbf{x}_i - \mathbf{x}_j\|^2}$, we will use a constant c to control the shape of the decision boundaries. However, since in the case of SVM we are interested in the effect of the size of the margin, we are fixing $c = \frac{1}{n}$ where n is the number of dimension of the samples. "One vs. Rest" method is used for SVM in this study.

1.5 Evaluation

In this study, we choose to evaluate three supervised learning algorithms. Furthermore, we derive and generalize these algorithms to perform multi-class classification. We extend these

algorithms using different kernel tricks, such as Polynomial and Gaussian kernels. To understand their generalization performance, we perform 5-fold cross validation on our evaluations. Finally, we evaluate and report the time complexity of these algorithms. For SVM only, the library 'sklearn' is used with parameters:
SVC(C=C, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=False, random_state=None, shrinking=True, tol=0.001, verbose=False)

We summarize our evaluations in Table 1.

Table 1: Experimental configurations

Config.	Algorithm	Kernel	Generalization method	Epochs	Cross validation
1	Perceptron	Polynomial	"One vs. Rest"	20	5-fold
2	Perceptron	Polynomial	"One vs. One"	20	5-fold
3	Perceptron	Gaussian	"One vs. Rest"	20	5-fold
4	kNN	Cosine similarity	"One vs. Rest"	20	5-fold
5	SVM	Gaussian	"One vs. Rest"	20	5-fold

1.5.1 Evaluating procedures

This study is divided into two parts. In the first part, we will understand the performance of different configurations listed in Table 1 by doing the followings:

1. For every hyperparameter θ in the parameter set S_θ.
   (a) Perform 20 runs:
      i. Random shuffle the input data and split it into 80% training set and 20% testing set.
      ii. Train the model using selected algorithm with hyperparameter θ and the training data set for 20 epochs (the reason for this number will be shown in the result section).
      iii. Perform and record the error rate of the model on the training and testing data set.
2. For all hyperparameter θ average over the training error and testing error. Compute their standard deviations.

We refer this procedure as the *basic procedure*.

In the second part, we move forward to find the best hyperparameter θ^* and evaluate the performance of different algorithms using θ^* on the testing data set, the flow of doing this is as follows:

1. Perform 20 runs:
   (a) Random shuffle the input data and split it into 80% training set and 20% testing set.

i. For every hyperparameter θ in the parameter set S_θ.

ii. 5 folds cross validation on hyperparameter θ by 20 epochs and record averaged validation accuracies.

(b) Select the best hyperparameter with the highest validation accuracy as θ^*.

(c) Use θ^* to test the algorithm on the testing data set.

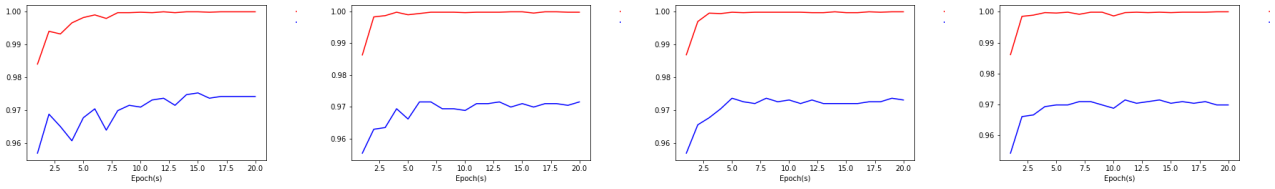
(d) Record θ^* and the testing error

2. Compute the mean and standard deviation of θ^* and testing error.

So in the second part, we will evaluate the algorithms more vigorously and find the optimal hyperparameter θ^* for this hand-written digits classification. We refer this procedure as the *cross validating procedure*.

1.5.2 Determining the number of epochs

To determine the number of epochs in the experiments, we performed a number of initial tests on the kernel peceptron algorithm with polynomial kernel and "One vs Rest" regime (Configuration 1 in Table 1). We found that the training accuracy stabilized after 20 epochs as shown in Figure 5. For this reason, we will use 20 epochs for all our experiments.



(a) $d = 3$, 1^{st} run (b) $d = 5$, 2^{st} run (c) $d = 6$, 9^{th} run (d) $d = 7$, 6^{th} run

Figure 5: Plots of training and testing accuracies using 20 epochs with polynomial kernel perceptron, where the x-axis is the number of epochs and the y-axis is the accuracy (%).

1.5.3 Cross validation

After we have trained the model, we still cannot guarantee it works well on unseen data. Therefore, we need to assure the accuracy of the predictions the model can achieve. So validation of the model is needed to evaluate the performance of the model by testing it on some unseen data. And then based on the performance on the validation data set, we can infer whether the model is under-fitting, over-fitting or just right. And for this, the technique named cross validation is used.

In this study, 5-fold cross validation is used to evaluate the performance of kernel perceptron, kNN and SVM on different parameters. Since the validation process is done 5 times, it results in a less biased model. We first split the training data set into 5 equal portions, then validate every fold after train the model using the remaining 4 folds of the training data. So all 5 folds are used in validating the performance of the model. The accuracy achieved in every fold is recorded and averaged. This is used as the performance measurement of the model.

Since we are using two kinds of kernels, polynomial and gaussian kernel for kernel perceptron, we are interested in exploring different d values from $1, ..., 7$ for the polynomial kernel $K_d(x_i, x_j) = (x_i \cdot x_j)^d$. We will first cross validate on these values and then choose the one gives the lowest validation error as the best parameter d^* for the polynomial kernel.

For the gaussian kernel $K_c(x_i, x_j) = e^{c\|x_i - x_j\|^2}$, since if x_i, x_j are very close together, the norm term $\|x_i - x_j\|^2 > 0$ will be small, so the gaussian kernel will give a value close to 1. When x_i, x_j are far apart, with similar reasoning, the gaussian kernel will give a value close to 0. We will test a range of values from 0 to 1 and then narrow down to a point where it gives the highest accuracy for kernel perceptron.

Apart from kernel perceptron, we will also cross validate on a range of values for the number of neighbors k in kNN to explore the effect of involving more neighbors for a prediction, and the penalty weight C in Kernel SVM to explore the effect of having different size of margins on its performance.

However, there will be some parameters remain not cross validated. The first one is the learning rate in the update rule of kernel perceptron. It is found that there can actually be a learning rate when we updating the weights in perceptron [10] while training. A larger learning rate may help us to find the optimal weight faster but it may also overshoot. A small learning rate may help prevent overshoot but the convergence time can be very slow. The second one is the number of epochs we used for kernel perceptron. In our study, we are just fixing the number of epochs as 20 because we saw that it can produce good and stable testing accuracy. However, this value may not be the optimal value such that the accuracy is maximized. Therefore, cross validating the number of epochs may find a value that help further improve the accuracy achieved by using 20 epochs.

1.5.4 Hyperparameter set

In this section, we will describe the hyperparameters that we are going to use in the experiment for different configurations. With reference to Table 1, for configuration 1 and 2, we will evaluate the performance of the algorithms using $d = \{1, 2, 3, 4, 5, 6, 7\}$. For configuration 3, the range of hyperparameter we are going to evaluate will be determined using the coarse to fine technique. Then for kNN in configuration 4, we will evaluate the algorithm using different number of neighbors $k = \{1, 7, 15, 31, 51, 101, 201\}$. Finally, for SVM, we will evaluate the algorithm by using different penalty weight $C = \{0.5, 1, 10, 50, 100, 500, 1000\}$. After this, we will cross validate on the hyperparameters and find the best hyperparameter using the technique discussed in Section 1.5.3. The mean and standard deviation of the hyperparameter and testing error for every configuration over the 20 runs will be shown. Exclusive for kernel perceptron, we will also look at the confusion matrix which we will mention in Section 1.5.5 and the five hardest to detect images mentioned in Section 1.5.6.

1.5.5 Confusion matrix

In this study, a confusion matrix is used to showcase the error rate of different combinations of predictions on the testing data set. Since we have ten classes, a 10×10 confusion matrix is used with the vertical dimension the testing data set true label (0 - 9) and the horizontal

dimension the predicted label (0 - 9). For example, if a '1' is mis-classified as '7', then a mistake is recorded on the entry $(1, 7)$. So every entry of the confusion matrix of every run is evaluating the following error rate:

$$\frac{\text{the number of times class } i \text{ mis-classified as class } j}{\text{total number of times class } i \text{ appeared in the testing data set}} \tag{51}$$

So after 20 runs, the 20 confusion matrices are averaged and the standard deviations of every entry over these matrices are calculated and will be presented as an overall 10×10 matrix as below:

$$\begin{bmatrix} \mu_{1,1} \pm \sigma_{1,1} & \dots & \mu_{1,10} \pm \sigma_{1,10} \\ \vdots & \ddots & \vdots \\ \mu_{10,1} \pm \sigma_{10,1} & \dots & \mu_{10,10} \pm \sigma_{10,10} \end{bmatrix} \tag{52}$$

where $\mu_{i,j}, \sigma_{i,j}$ are mean and standard deviation of the error (51) that class i mis-classified as class j respectively.

1.5.6 Evaluating the five hardest to detect images

In this study, we will also showcase the five hardest to detect samples' images, these images are determined by the 20 best predictors (model using the best hyperparameter θ^*) while performing the 20 runs. These 5 images are defined as the 5 most frequent wrongly predicted samples using these 20 optimized models.

This will show us some insights about the decision boundaries and this will be further discussed in the Section 1.6.

1.6 Results and Discussion

In this section, we will first look at the result from kernel perceptron. Then we will look at the result from kNN. Lastly, we will look at the result from SVM.

1.6.1 Kernel perceptron results

Basic results In this part we will evaluate on the training and testing errors on kernel perceptron using different hyperparameters on polynomial and gaussian kernel using basic procedure mentioned in Section 1.5.1.

Table 2: Training and Testing error (%) of polynomial kernel perceptron using "One vs Rest" with different kernel dimension (d).

$d =$	Training error (%)	Testing error (%)
1	5.885 ± 1.249	8.790 ± 1.072
2	0.054 ± 0.037	3.019 ± 0.377
3	0.019 ± 0.011	2.696 ± 0.274
4	0.034 ± 0.061	2.629 ± 0.263
5	0.073 ± 0.279	2.919 ± 0.465
6	0.057 ± 0.149	2.691 ± 0.286
7	0.043 ± 0.073	2.839 ± 0.329

Table 3: Training and Testing error (%) of polynomial kernel perceptron using "One vs One" with different kernel dimension (d).

$d =$	Training error (%)	Testing error (%)
1	1.7283 ± 0.6913	5.9328 ± 0.7222
2	0.0296 ± 0.0243	3.3145 ± 0.3401
3	0.0497 ± 0.1217	3.2849 ± 0.4427
4	0.0323 ± 0.0502	3.2204 ± 0.4016
5	0.0235 ± 0.0348	3.3441 ± 0.3864
6	0.1586 ± 0.4408	3.4812 ± 0.3820
7	0.0134 ± 0.0190	3.6183 ± 0.3481

Polynomial kernel - "One vs. Rest" We report the training and testing error of polynomial kernel perceptron using "one vs Rest" in Table 2. It is clear that all the errors are very small except when $d = 1$. Starting from $d = 2$, both errors go down slightly and then increase again. Training error reaches its smallest error at $d = 3$ and testing error reaches its smallest error at $d = 4$. This shows the non-linear nature of the data set because when $d = 1$ the error of both training and testing error is the highest. One possibility is that the decision boundaries at $d = 1$ (linear) lies between the classes but with more data points on the wrong side (underfitting) compared to those when $d \neq 1$. Start from $d = 2$ the decision boundaries are bent (not a straight line), so they are more flexible to fit the data. However, when the boundaries bent too much some data will go to the wrong side of the boundaries again because of overfitting.

Polynomial kernel - "One vs One" The performance of kernel perceptron using polynomial with the same set of d using the generalization technique "One vs. One" is shown in Table 3. One observation in Table 3 is that the training and testing error for $d = 1$ is smaller compared to that when we are using "One vs. Rest". This is because when we are only dealing with two classes, the decision boundaries are dealing with much less data, meaning that it is much easier to place the decision boundaries between the two classes because data from all other classes are gone. Therefore, there is less data on the false side compared to when using "One vs. Rest". This improve the performance for testing because the sample will have a higher chance of falling into the correct side of the decision boundaries. However, this is for the case when $d = 1$ only.

Although we can see that the training error is generally less then that using "One vs. Rest" when $d > 1$, the testing error increased. The reason is due to the bent nature of the decision boundaries and the much less quantity of data we are dealing with in the "One vs. One" case. In this setting, $d > 1$ decision boundaries can easily overfit for the data when training. So the model will be more biased to the training data in a way that it will also capture the noisy data. This is traded off by higher testing errors because when the model is overfitted for the training data, the testing data has a higher chance to fall on wrong side of the nonlinear boundaries.

Gaussian kernel - "One vs Rest For gaussian kernel, different values between 0 and 1 are tested with 10% of the data with the "One vs. Rest" generalizing method to find the best

value of c that can produce the highest accuracy. We will first coarse search for such value by finding the minimum testing error when using different values of c in Table 4.

Table 4: Coarse parameter search results ....

$c =$	Testing error (%)
0.000001	69.2204
0.00001	52.4462
0.0001	20.4570
0.001	8.0914
0.01	6.3710
0.001	9.059
0.005	5.7258
0.01	5.5645
0.05	9.6237
0.1	11.5323

Table 5: Fine parameter search results ....

$c =$	Testing error (%)
0.005	6.2097
0.006	6.6129
0.007	5.8871
0.008	5.6452
0.009	6.1022
0.01	6.1022
0.02	6.3172
0.03	7.284
0.04	8.4140
0.05	9.4624

From Table 4 we can see that the lowest testing error rate is around 0.001 and 0.1, so we are going to fine search values from 0.005 to 0.05 as shown in Table 5. In Table 5 we can see that the minimum testing error is around $c = 0.008$. We will now further search around the range around $c = 0.008$ from 0.007 to 0.009 with a step size of 0.0002 with the full data set as shown in Table 6

Table 6: Training and Testing error (%) of gaussian kernel perceptron using "One vs. Rest" with different values of c

$c =$	Training error (%)	Testing error (%)
0.0070	0.0168 ± 0.0164	2.8226 ± 0.4117
0.0072	0.0134 ± 0.0104	2.6613 ± 0.4579
0.0074	0.0208 ± 0.0138	2.7151 ± 0.3290
0.0076	0.0229 ± 0.0165	2.6398 ± 0.3558
0.0078	0.0168 ± 0.0119	2.6210 ± 0.4005
0.0080	0.0202 ± 0.0202	2.5726 ± 0.3963
0.0082	0.0457 ± 0.0749	2.7124 ± 0.3428
0.0084	0.0235 ± 0.0387	2.6398 ± 0.3171
0.0086	0.0182 ± 0.0155	2.6398 ± 0.2210
0.0088	0.0215 ± 0.0249	2.5806 ± 0.3116
0.0090	0.0229 ± 0.0332	2.7016 ± 0.3813

We can see that in Table 6 the gaussian kernel is doing a better prediction comparing to the polynomial kernel in Table 2. The training error and testing error is generally lower than that of polynomial kernel. This may due that the gaussian kernel has an infinite number of dimensions in the feature space since it can be expanded using Taylor Series. Compared to gaussian kernel, polynomial kernel just have $\binom{n+d}{d}$ dimensions in the feature space, so the gaussian kernel is much more flexible than polynomial kernel. Therefore, gaussian kernel can give a better performance than polynomial kernel. However computing gaussian kernel takes more time.

Best hyperparameters of polynomial and Gaussian kernel After cross validating the hyperparameters, the mean and standard deviation of the optimized hyperparameters (d^*, c^*) and the testing error is as in Table 7:

Table 7: Mean and standard deviation of the optimized hyperparameters and the testing error

	Polynomial kernel with "One vs. Rest"	Polynomial kernel with "One vs. One"	Gaussian kernel with "One vs. Rest"
Testing error (%)	3.1048 ± 0.4841	3.2608 ± 0.3679	2.7043 ± 0.3859
Optimal hyperparameter	$d^* = 4.3 \pm 1.0536$	$d^* = 3.6 \pm 1.1136$	$c^* = 0.00814 \pm 0.0006$

In Table 7, we can observe that the testing error of the "One vs. Rest" case is smaller than that of "One vs. One" case using the best hyperparameter. This is because the decision boundaries fit better to the data in "One vs. One" case since it has much less data to deal with when training. So it is more biased to the training data and result in a higher error in the test data.

Moreover, we can also see that the optimal hyperparameter d^* is smaller when using "One vs. One", the reason we result in a lower d^* is because we are using the validation data to determine the optimal hyperparameter. Since the validation data set is actually part of the data set we used for training and all the training data actually had a chance to be in the validation data set, the model is actually validating on the training data set. Also, the model can overfit the validation data easier using "One vs. One" technique with a smaller d because we have much smaller data set when doing validation. With the above two reasons, we can conclude the reason for a smaller optimal hyperparameter for "One vs. One".

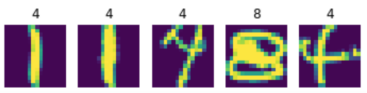
Confusion matrix In this section, we explore the error rate of different combinations of predictions on the testing data set when using polynomial kernel with "One vs. Rest" generalization method. A 10×10 confusion matrix is constructed below with each entry presenting the mean $\mu_{i,j}$ and standard deviation $\sigma_{i,j}$ (in percentage) of the error (51) that class i mis-classified as class j in the form of $\mu_{i,j} \pm \sigma_{i,j}$:

$$\begin{bmatrix}
0 \pm 0 & 0.0642 \pm 0.1284 & 0.2564 \pm 0.2155 & 0.0943 \pm 0.2226 & 0.1267 \pm 0.1554 & 0.1902 \pm 0.2329 & 0.3834 \pm 0.3482 & 0.0317 \pm 0.0950 & 0.0327 \pm 0.0983 & 0.0958 \pm 0.1465 \\
0.0207 \pm 0.0901 & 0 \pm 0 & 0.0181 \pm 0.0787 & 0.0211 \pm 0.0920 & 0.1795 \pm 0.266 & 0 \pm 0 & 0.2819 \pm 0.3622 & 0 \pm 0 & 0.1543 \pm 0.2211 & 0.0395 \pm 0.1186 \\
0.4863 \pm 0.6801 & 0.2391 \pm 0.3068 & 0 \pm 0 & 0.5355 \pm 0.5277 & 0.9097 \pm 0.5884 & 0.1342 \pm 0.2903 & 0.2087 \pm 0.4182 & 0.5628 \pm 0.5971 & 0.4464 \pm 0.5447 & 0.0534 \pm 0.1601 \\
0.3593 \pm 0.4685 & 0.2712 \pm 0.3989 & 0.6114 \pm 0.7649 & 0 \pm 0 & 0 \pm 0 & 1.9766 \pm 0.9066 & 0.0307 \pm 0.1337 & 0.5032 \pm 0.5732 & 1.2440 \pm 0.8735 & 0.0287 \pm 0.1253 \\
0.0570 \pm 0.1719 & 0.7296 \pm 0.5086 & 0.7690 \pm 0.6753 & 0.0523 \pm 0.1574 & 0 \pm 0 & 0.1776 \pm 0.2722 & 0.7224 \pm 0.6355 & 0.2470 \pm 0.3594 & 0.1712 \pm 0.3666 & 1.1374 \pm 0.6791 \\
0.7932 \pm 0.7247 & 0.1336 \pm 0.3403 & 0.4464 \pm 0.6036 & 0.8907 \pm 0.6984 & 0.4488 \pm 0.5461 & 0 \pm 0 & 1.1898 \pm 0.9932 & 0.0686 \pm 0.2059 & 0.4565 \pm 0.4044 & 0.4902 \pm 0.5108 \\
1.0832 \pm 0.9295 & 0.2397 \pm 0.3924 & 0.2964 \pm 0.4793 & 0 \pm 0 & 0.3303 \pm 0.4433 & 0.1799 \pm 0.3310 & 0 \pm 0 & 0 \pm 0 & 0.3968 \pm 0.4493 & 0.0273 \pm 0.1191 \\
0 \pm 0 & 0.3465 \pm 0.5170 & 0.4555 \pm 0.5859 & 0.0602 \pm 0.1811 & 0.9445 \pm 0.6384 & 0.0615 \pm 0.1849 & 0 \pm 0 & 0 \pm 0 & 0.5517 \pm 0.6918 & 0.7929 \pm 0.7768 \\
0.9279 \pm 0.8790 & 0.2879 \pm 0.4775 & 0.5467 \pm 0.5593 & 1.0859 \pm 0.7633 & 0.4622 \pm 0.5323 & 1.1329 \pm 0.7960 & 0.5012 \pm 0.3972 & 0.5737 \pm 0.5527 & 0 \pm 0 & 0.2570 \pm 0.4909 \\
0.2110 \pm 0.3429 & 0.1166 \pm 0.2989 & 0.2101 \pm 0.2869 & 0.0906 \pm 0.2162 & 1.1747 \pm 0.7964 & 0.1823 \pm 0.2793 & 0.0296 \pm 0.1290 & 1.0459 \pm 0.8294 & 0.1589 \pm 0.3339 & 0 \pm 0
\end{bmatrix}$$

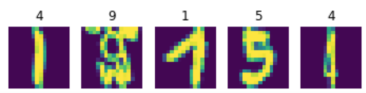
We can see that there are some digits that are mis-classified more frequently than others. For example the digit "8" is mis-classified as a range of other digits including "0", "2", "3", "5", "6", "7", all with an error rate $> 0.5\%$.

Five hardest to detect images In this section, the five hardest to detect images when using "One vs. Rest" with polynomial kernel, "One vs. One" with polynomial kernel and "One vs. Rest" with gaussian kernel is presented in Figure 6.

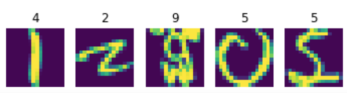
In most of the cases, it is not surprising that these digits are hard to predict because of two reasons.



(a) Polynomial - "One vs Rest"



(b) Polynomial - "One vs One"



(c) Gaussian - "One vs Rest"

Figure 6: The images that are most difficult to predict, listed with respect to their kernel trick and the generalization method.

Firstly, we can see that some of the images have the wrong labels. For example, the first image in Figure 6a, 6b, and 6c is clear a "1" but mis-labeled as "4".

Secondly, there are noisy images such as the second image in Figure 6a which looks like a poorly drawn mickey mouse, and the fourth image in Figure 6b which is a poorly written "8". Apart from the noisy images, there are also digits written in a way that looks like another digit. Instances can be found in the third images of Figure (6b) and (6a). Although they seems to have the correct label, they all looks like a "7".

So there are two reasons in total that can result in mis-classification.

However, there are still some images that should be correctly classified appeared in this list. For example, the fifth image in Figure (6b) and the fourth image in Figure (6a), which are digit "4" and "5" respectively. These mis-classification means that the resultant perceptrons may tends to lean more on some class of the training samples, resulting in this kind of mis-classification. We will later on see that in SVM, since we will try to find the boundary that in right in the middle between two classes, we will not see this kind of mis-classification appear in this list.

1.6.2 k-Nearest Neighbors (kNN) results

Basic results In this part we will evaluate the training and testing errors on kNN using different number of neighbors k using the basic procedure mentioned in Section 1.5.1 to explore the effect of involving more neighbors for a prediction. The result is presented as below:

$k =$	Training error (%)	Testing error (%)
1	0 ± 0	3.5188 ± 0.4096
7	3.1689 ± 0.1102	4.3656 ± 0.3441
15	4.5839 ± 0.1263	5.4355 ± 0.4466
31	6.5407 ± 0.1329	7.1667 ± 0.4650
51	8.3127 ± 0.1484	8.6613 ± 0.7459
101	11.1522 ± 0.1710	11.2151 ± 0.5313
201	15.2810 ± 0.1592	15.5054 ± 0.7922

From the above data, we can see that kNN is doing an excellent job on the training error when $k = 1$. This is because while predicting on the training data, every data will always find themselves so there will always be a cosine similarity value $cos\theta = 1$.

Apart from the above, we can also notice that as the number of neighbors increase, the training and testing error also increased. This is because as we increase k, we will have more bias on

the training data set so the training error will increase. Similar reasoning to that of the testing data, as we increase k we are more biased to the training data so we have an increasing trend of testing error.

Compared to kernel perceptron and SVM, instead of having a minimum training and testing error at certain hyperparameter, kNN has its training and testing error increase monotonically as k increases. Since kNN is just directly comparing the testing sample with the training samples, it require less steps compare to kernel perceptron and SVM, along with the vectorization done in the programme, it run the fastest among the three algorithms.

Best number of neighbors Below shows the resultant mean and standard deviation of testing error and the optimal hyperparameter k^* by using the cross validating procedure mentioned in Section 1.5.1:

- Mean and standard deviation of testing error: $3.5242 \pm 0.3588\%$
- Mean and standard deviation of k^*: 1.0 ± 0.0

As we can see from the basic results, kNN is doing the best in the training data set when $k = 1$, so it is not surprising that the optimal $k^* = 1$ after doing the 5 folds cross validation. the testing error also agree with that in the basic results.

Five hardest to detect images This part shows the hardest to detect images in the case of kNN with cosine similarity.

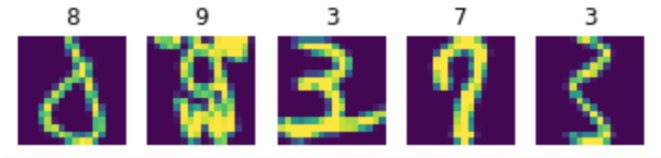


Figure 7: Five hardest to detect images and their true labels when using "One vs. Rest" method and cosine similarity in kNN

This is an interesting result since as mentioned in Section 1.5.1, these samples were found by 20 models using the best hyperparameters. As mentioned in the kNN part in Section 1.2.2, kNN is very sensitive to noisy data and there is no concrete training for kNN, this algorithm is just like a direct comparison of the testing sample data with all the training samples. In Figure 7, we can see how kNN is sensitive to noisy data. Although it is reasonable for the algorithm to have the first (a "0" like "8") and the second image (mickey mouse like "9") mis-classified, the last three images should be some easy to classify digits because they are just slightly noisy. The third image has horizontal line at the bottom of the digit "3" and the forth and fifth images show slightly distorted "7" and "3". So, compared to kernel perceptron and kernel SVM, kNN is less robust. This also shows the advantages of models with proper training as in kernel perceptron and SVM because this kind of mis-classification happen less in these two algorithms, especially in SVM.

1.6.3 SVM results

Basic results In this part we will evaluate the training and testing error using different penalty weight C which controls the width of the margin using the basic procedure in Section 1.5.1. The result recorded is as shown in Figure 8.

Table 8: Training and testing error (%) of SVM with Gaussian kernel, varying panelty weight C.

$C =$	Training error (%)	Testing error (%)
0.5	2.5934 ± 0.0988	3.9220 ± 0.5007
1	1.6792 ± 0.0752	2.9866 ± 0.4448
10	0.1042 ± 0.0147	2.2016 ± 0.3142
50	0.0242 ± 0.0054	2.3333 ± 0.3192
100	0.0175 ± 0.0105	2.4462 ± 0.3119
500	0.0101 ± 0.0058	2.4140 ± 0.3162
1000	0.0054 ± 0.0066	2.3925 ± 0.2775

As we mentioned in the SVM part in Section 1.2.2, when C is small, the weight of mis-classification is low. So the margin is large so it allows more data to be in the margin or even on the wrong side of the margin, resulting in a model with high bias and low variance. When C is large, since the weight of mis-classification is high, this force the SVM algorithm to fit as many training data as possible so it may overfit. So a large penalty weight C will result in a model with high variance and low bias.

From the above table, we can see that the training error decrease as the value of C increase. This shows sign of overfitting at very large value of C. For the testing error, we can see that when C increase, the testing error first decrease to its minimum ($2.2016 \pm 0.3142\%$) when $C = 10$ and then increase slightly as C further increase. That mean for this data set of hand written digits, the bias and variance is at their minimum when C is around 10.

Compared to the result in kernel perceptron when using gaussian kernel, we have a testing error of $2.7043 \pm 0.3859\%$ when it was using it's best predictor (with $c^* = 0.00814 \pm 0.0006$). We can see that in the case of SVM, we can get a even lower testing error by tuning the width of margin. It seems that tuning the margin is a more effective way if we want to get a higher testing accuracy.

Best penalty weight In this part the result of using the procedure of the second part mentioned in beginning of section 1.3 is evaluated in terms of mean and standard deviation of the testing error and the optimal hyperparameter C^*:

- Mean and standard deviation of testing error: $2.2715 \pm 0.3185\%$
- Mean and standard deviation of C^*: 14.0 ± 12.0

Compared to the result of kernel perceptron when using gaussian kernel, SVM give a lower testing error. This reinforce what we mentioned in the basic result that tuning the margin is a more effective way to get a higher testing accuracy.

Five hardest to detect images This section shows the five hardest to detect images in the case of SVM using "One vs. Rest" method and gaussian kernel.

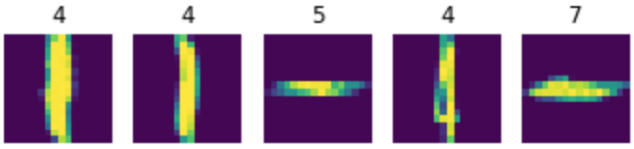


Figure 8: Five hardest to detect images and their true labels when using "One vs. Rest" method and gaussian kernel in kernel SVM

From Figure (8), we can see that in SVM it is not surprising to have all these images mis-classified because they are all mis-labeled digits. Compare to kernel perceptron, we can see that introducing a margin between different classes for the decision boundaries does help improve the the performance. This is improved in the way that the decision boundaries are now places around the center between the classes, so they are not leaned more on some classes of training samples. With this, the mis-classification mentioned in Section 1.5.6 is not happening here in kernel SVM.

1.6.4 Computational time complexity

In this section, we explore the time complexity of the three algorithm we evaluated. First we define n as the number of training sample, p the number of features and k as the number of classifiers. Since only the "One vs. Rest" generalization method is used across all algorithms, we will just compare configuration 1, 4 and 5 in Table 1 here.

Time complexity of kernel perceptron In the program, we first precalculate the kernel matrix which is expressed in the form (36), the kernel computation has time complexity $\mathcal{O}(pn^2)$, then we loop through all n training samples and for all training samples, there will be a matrix multiplication of the α matrix sized $(k \times n)$, and an $(n \times 1)$ kernel vector. So the time complexity will be $\mathcal{O}(kn^2)$ for training. In the case of testing, there will be just one matrix multiplication between the $(k \times n)$ α matrix and an $(n \times 1)$ kernel vector, so the time complexity is $\mathcal{O}(kn)$. Moreover, k is just a constant, so the time complexity is be further reduced to $\mathcal{O}(n^2)$ for training and $\mathcal{O}(n)$ for testing.

Time complexity of k-Nearest Neighbors (kNN) There is no training time complexity for kNN as it is just direct comparison between the testing sample and the n training samples, with each of the samples having p features. So the testing time complexity is $\mathcal{O}(np)$.

Time complexity of SVM Since the gaussian kernel expressed in (40) is evaluated with a sum of matrix multiplication, the kernel computation has $\mathcal{O}(pn^2)$ time complexity and the quadratic programming problem needs $\mathcal{O}(n^3)$ time complexity, so the time complexity for training SVM is $\mathcal{O}(n^2p + n^3)$. For testing, since after training the SVM will only depends on

its support vectors and the testing sample has p features, SVM has $\mathcal{O}(pn_{SV})$ time complexity where n_{SV} is the number of support vectors.

It seems that from the theoretical point of view, kNN is the fastest, then kernel perceptron comes next and the slowest is SVM. A reason why kernel perceptron is relatively faster may due to the fact that kernel perceptron will only update when there is a mistake. To verify, their average training time over 20 epochs are calculated as below:

- Kernel perceptron (Configuration 1): 4.5280 seconds/epoch
- kNN (Configuration 4): 2.4416 seconds/epoch
- SVM (Configuration 5): 5.9096 seconds/epoch

So the experimental data agree with the theoretical value.

1.7 Conclusion

In this study, the theory of three algorithms: Kernel perceptron, kNN and SVM with gaussian kernel were reviewed. Next, with the hand-written digit data, training and testing errors are computed for a range of hyperparameters, then the hyperparameters are cross-validated. After that, the five hardest to detect images for all the algorithms are presented along with their true labels. Exclusive for kernel perceptron (Configuration 1), a 10×10 confusion matrix is presented to showcase the error rate of different combinations of predictions on the testing data set.

Supported by the evaluation, SVM is found to be the best algorithm in terms of accuracy and the five hardest to detect images generated by SVM is the best in the sense that it is not surprising to have all the images mis-classified because they were all mis-labelled. However, SVM is the worse performing algorithm when it comes to time. This is because both theory and experimental data suggested that SVM is the slowest performing candidate among the three chosen algorithm. The effect of adjusting the penalty weight on the accuracy is also explored, we can see that tuning the penalty weight may be a more effective way if we want a higher accuracy, and the resultant decision boundaries will be more likely to be placed right in the middle between the two classes of data.

Kernel perceptron is the second best in terms of both the accuracy and speed. In contrast to SVM, kernel perceptron trains one sample at a time so it is an online learning algorithm. Two different generalization method, namely "One vs. Rest", "One vs. One" and two types of kernels, polynomial and gaussian kernel, are evaluated using kernel perceptron. We can see that in the best cases, "One vs. Rest" has a smaller testing error compared to that of "One vs. One" because of the higher bias of the "One vs. One" decision boundaries to the training data. We also noticed that gaussian kernel can out perform polynomial kernel when comparing in best case. This may due to the fact that gaussian boundaries has a more flexible nature when compared to polynomial boundaries which are restricted to certain shapes.

kNN is the fastest among the three algorithms because kNN is just performing a direct comparison between the training data and the testing sample and then choose the most frequent class

from a group of k closest neighbors. However, the accuracy is the worst in this study because kNN is very sensitive to noisy data. We can see that from the worst five images displayed in Figure 7 that three out of five images in the figure are just slightly distorted images.

In general, the objective of this study were achieved. This study has proven the possibility of superivised learning algorithms on multiclass classification problems. Although multiclass classifying hand written digits seems a trivial problem for human, it opens up many opportunities in the machine world.

Part II

This question consider the sample complexity of 4 algorithms

1. Winnow
2. Perceptron
3. Least squares
4. 1-nearest neighbours (1NN)

for the problem 'just a little bit', which have m samples with each of them having n dimensions/features. Each feature is sampled uniformly from $\{-1, 1\}$ and the true label y of each sample is the first entry of the sample x. To illustrate:

$$X = \begin{bmatrix} 1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix}, Y = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}$$

So the true labels are just the first column and all others have nothing to do with it. In this question, we are interested in the sample complexities for the above 4 algorithms in the sense of the minimum number of samples (m) to get $\leq 10\%$ generalization error on average. The generalization error for each sample is defined as follows:

$$\varepsilon(\mathcal{A}_{\mathcal{S}}) := 2^{-n} \sum_{x \in \{-1,1\}^n} I[\mathcal{A}_{\mathcal{S}}(\mathbf{x}) \neq x_1] \tag{53}$$

We can see that from the formula, what it is doing is to count all the mistakes made by the algorithm $\mathcal{A}_{\mathcal{S}}$ trained from data sequence on the sample for all combination that a sample spanning $\mathbb{R}^n$ can have. So there are 2^n combinations in total. In a nutshell, this boils down to a simple fraction:

$$\frac{\text{number of mistakes made by the algorithm } \mathcal{A}_{\mathcal{S}} \text{ when predicting all the combinations}}{\text{total number of combinations that a sample spanning n dimensions}}$$

Therefore, with (53) the sample complexity on average at 10% generalization error $\mathcal{C}(\mathcal{A})$ is:

$$\mathcal{C}(\mathcal{A}) := \min\{m \in \{1, 2, ...\} : \mathbb{E}[\varepsilon(\mathcal{A}_{\mathcal{S}})] \leq 0.1\} \tag{54}$$

which means the minimum number of sample (m) needed for the algorithm $\mathcal{A}_{\mathcal{S}}$ to get a generalization error $\leq 10\%$ averaging over m samples.

(a) In this section, the 4 algorithms are implemented and plots of estimated sample complexity (m) over sample dimension (n) are generated with standard deviation bars:

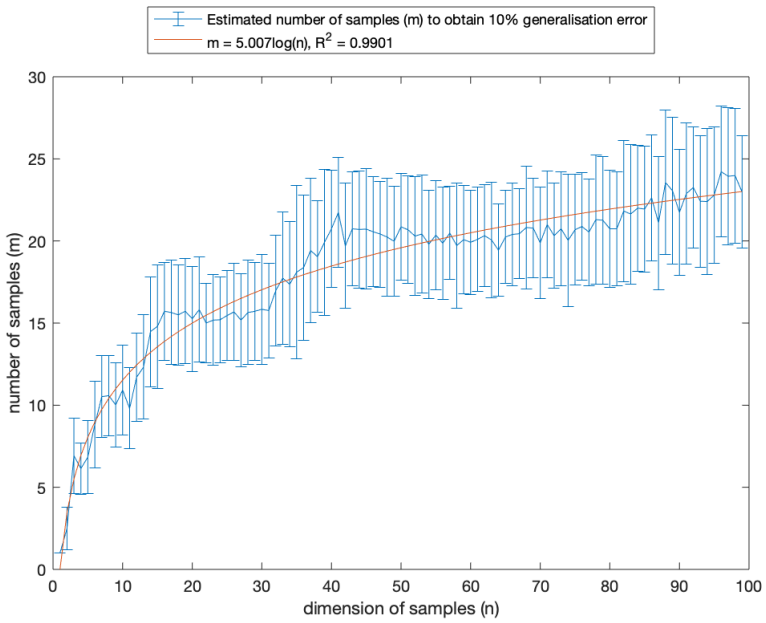


Figure 9: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for winnow

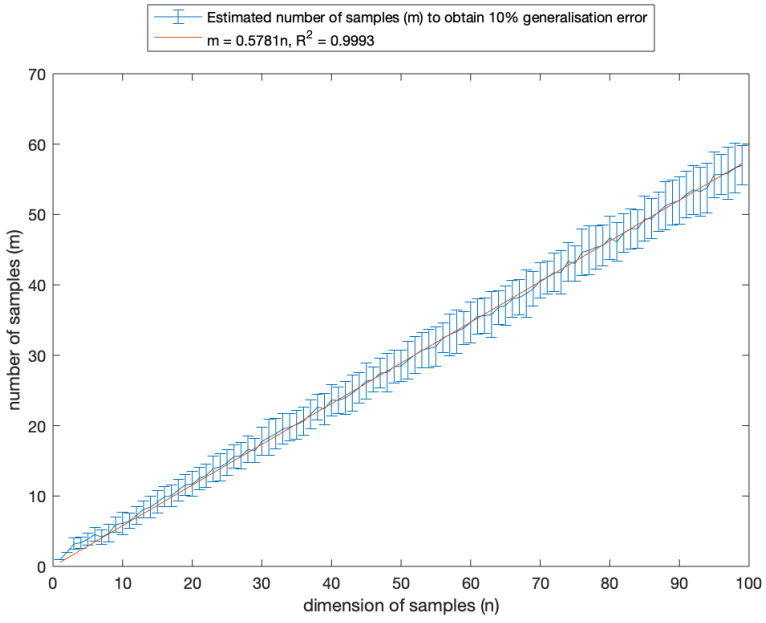


Figure 10: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for least square

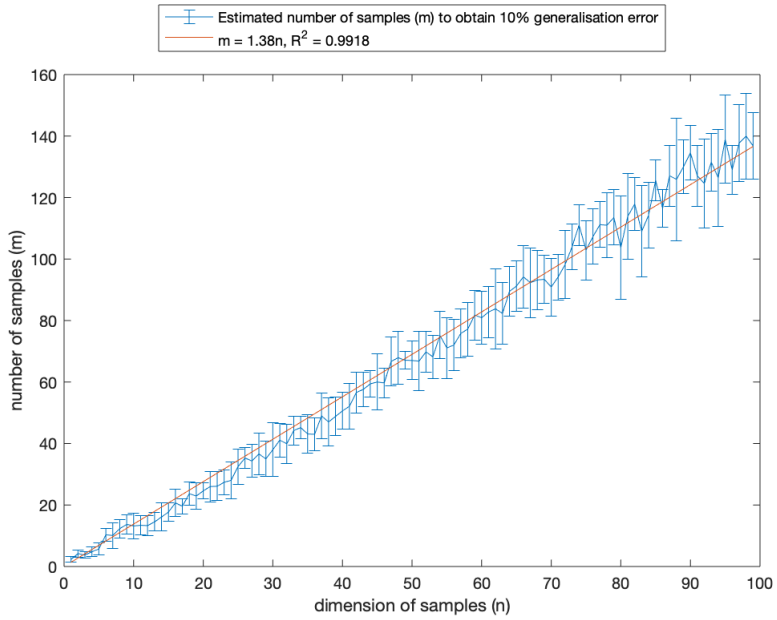


Figure 11: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for perceptron

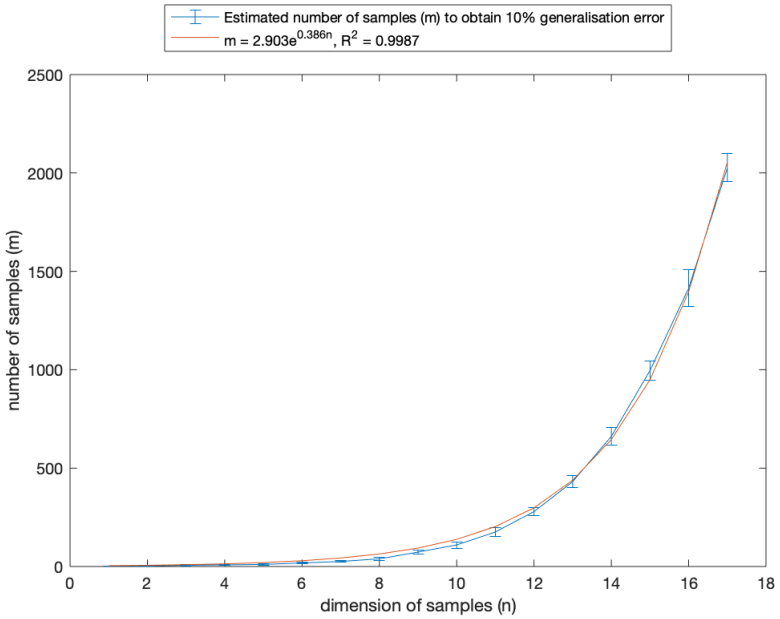


Figure 12: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for 1NN

Since as n increase, the computation time for 1NN becomes exponentially large, 1NN is tested up to dimension $n = 17$.

(b) The program have done four things to approximate generalization error:

(a) Test algorithms for limited number of sample $1 \leq m \leq 10000$ and dimension $1 \leq n \leq 100$.

(b) Random generate m training samples with n dimensions.

(c) Random generate 10000 testing samples with n dimensions.

(d) Trial for 10 times, escape every trial whenever the program found an m that can have a testing error ≤ 0.1

Since we do not have all combinations of data, the generalization error is reduced to testing error:

$$\frac{1}{m_{test}} \sum_{i \in S_{test}} I[\hat{y}_i \neq y_i] \tag{55}$$

Trade-offs

(a) Since we generate training and test datasets randomly, the combination generated for them may not include all the combinations of data needed so we gave up some accuracy for the generalization error for computational time.

(b) As the number of dimension n increase, the validation data generated will less likely to cover all the 2^n combination of samples, so as n increase, the accuracy will decrease. For example, when $n = 10$ there are $2^{10} = 1024$ combinations, so generating 10000 testing samples have a high chance to cover all the combinations we need. However, when $n = 30$ there will be 2^{30} combinations and will exceed 10000, so we are getting a smaller and smaller subset of the full combination of samples. Therefore, accuracy is traded-off for computational time.

Bias
In my case, my estimated sample complexity is always less that that of the true sample complexity. This is because for every trial we test m from 1 to 10000, there will be a tendency of getting a testing error ≤ 0.1 at a value m lower than that for the true value.

(c) As in the figures presented in a), we can see that for the 'just a little bit' problem, winnow has the best performance such that as the dimension of the samples increases, the minimum number of sample needed to achieve 10% error does not increase as much. By observing the trend, it seems that the sample complexity grows logarithmically as a function of dimension.

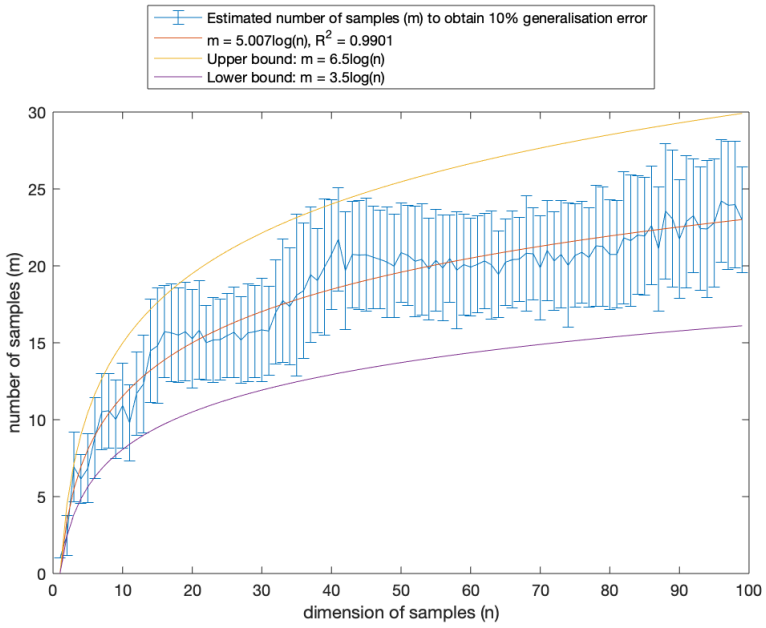


Figure 13: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for winnow with upper and lower bounds

By curve fitting, it is found out that:

$$m = 5.007log(n), R^2 = 0.9901$$

Since we can see that the upper bound and lower bound also grows logarithmically, we can conclude that $m = \Theta(log(n))$ for winnow.

The second best algorithm for this problem is least square since the trend grows in a linear manner, the minimum number of sample (m) needed will exceed that of winnow at some point as the dimension (n) increases.

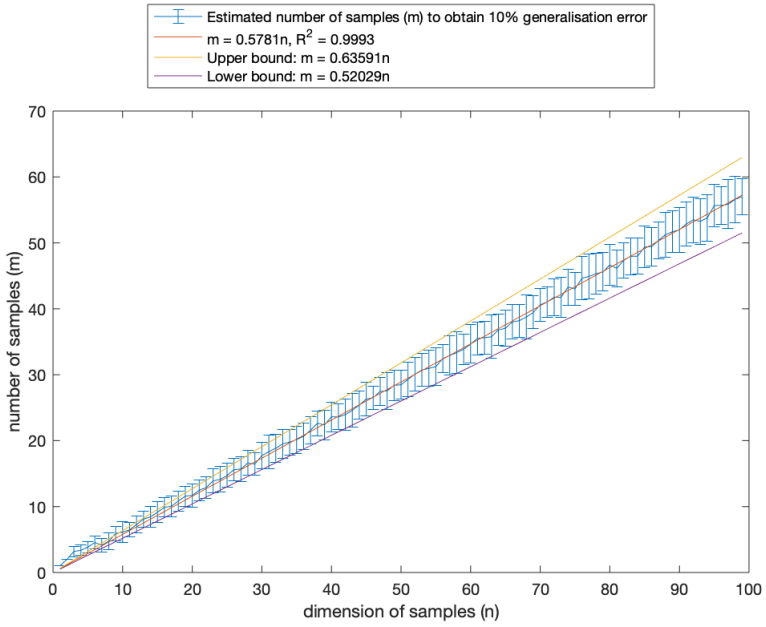


Figure 14: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for least square with upper and lower bounds

By curve fitting it is found out that:

$$m = 0.5781n, R^2 = 0.9993$$

After least square is perceptron, which also has the trend grows in a linear manner but with a higher rate.

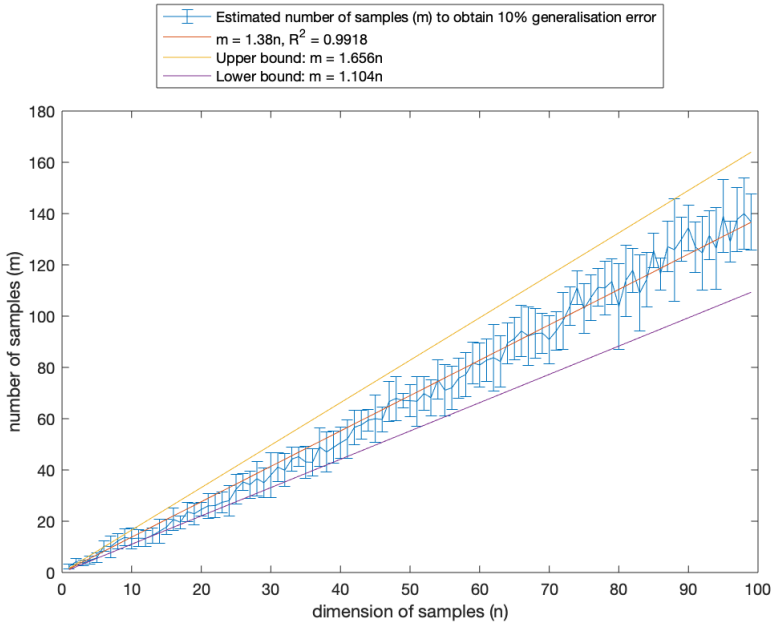


Figure 15: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for perceptron with upper and lower bounds

By curve fitting it is found out that:

$$m = 1.38n, R^2 = 0.9918$$

For least square and perceptron, since the upper bound and lower bound grows linearly we can conclude that for least square and perceptron, $m = \Theta(n)$

The worst performed algorithm is 1NN since its trends grows in an exponential manner.

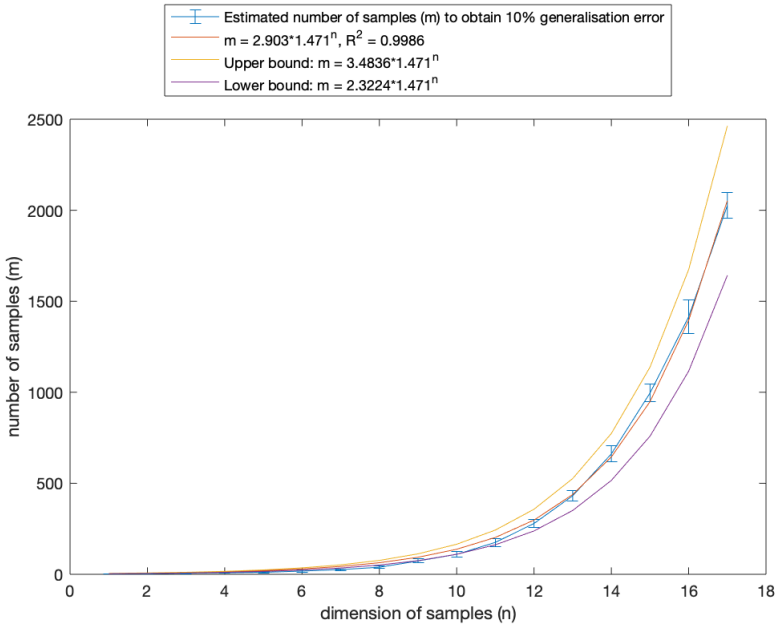


Figure 16: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for 1NN with upper and lower bounds

By curve fitting it is found out that:

$$m = 2.903 \times 1.471^n, R^2 = 0.9986$$

In this case the upper bound and the lower bound for 1NN also grows exponentially in similar manner as that of the mean, so $m = \Theta(1.471^n)$.

This means as the sample dimension (n) grows, 1NN will become so data-hungry that it needs more samples to achieve a 10% error. This burden in practical sense since we usually have to work under limited resources of data.

(d) Suppose the data is drawn from a distribution P and a mistake bound for algorithm $\mathcal{A}$ for any such set is B. We can use Batch Bound Theorem: Given m, then let s be drawn uniformly at random from $\{1, ..., m\}$. Let $\mathcal{S}$ be consist of $s-1$ examples sampled i.i.d from P, let $(\mathbf{x}_s, y_s)$ be an additional example sampled from P then:

$$P[\mathcal{A}_{\mathcal{S}}(\mathbf{x}_s) \neq y_s] \leq \frac{B}{m} \tag{56}$$

with respect to the draw of s, $\mathcal{S}$ and $(\mathbf{x}_s, y_s)$.

For perceptron, the mistake bound we have is the Novikoff Bound:

$$B \leq \left(\frac{R}{\gamma}\right)^2 \tag{57}$$

Here $R := \max_i \|\mathbf{x}_i\|$, and the margin $\gamma \leq y_i(\mathbf{v} \cdot \mathbf{x}_i)$ where $\mathbf{v}$ is a vector with $\|\mathbf{v}\| = 1$.

In our case, since all our feature can just be either -1 or $+1$ and with the constraint $\gamma \leq y_i(\mathbf{v} \cdot \mathbf{x}_i)$ actually implies the perceptron will go through the origin so the maximized margin γ is of magnitude 1.

Also, because of the -1 or $+1$ values for all the features, we will have maximum value of $R := \max_i \|\mathbf{x}_i\| = \sqrt{n}$ when all entries in $\mathbf{x}_i$ is 1. So combining the two arguments, we will have the mistake bound ():

$$\begin{aligned} B &\leq \left(\frac{\sqrt{n}}{1}\right)^2 \\ &\leq n \end{aligned} \tag{58}$$

Now we substitute (58) back to (56) we get:

$$P[\mathcal{A}_{\mathcal{S}}(\mathbf{x}_s) \neq y_s] \leq \frac{n}{m} \tag{59}$$

So the upper bound $\hat{p}_{m,n}$ on the probability that the perceptron will make a mistake on the s^{th} example is:

$$\hat{p}_{m,n} = \frac{n}{m} \tag{60}$$

(e) As the number of dimension n increase, the number of samples m need to increase accordingly. However, for 1NN as we can see in the part 2c), as n increase, m increase exponentially, which means as the number of dimension increase in 1NN case, the algorithm become more and more hungry for data. This is a phenomenon of the curse of dimensionality. So:

$$m = \Theta(1.471^n) \tag{61}$$

Now, what we also know is that we have our data is randomly generated. If the data is not randomly generated, the data may biased to one corner of the dimension space which may need much more sample complexity because the algorithm struggle to predict the correct label when the data set is biased. Therefore, randomly generating data is actually the best case the algorithm can have because the data is uniformly distributed across the dimension space. Since the lower bound implies the best case, we have:

$$m = \Omega(1.471^n) \tag{62}$$

References

[1] F. Noé, G. De Fabritiis, and C. Clementi, "Machine learning for protein folding and dynamics," *Current Opinion in Structural Biology*, vol. 60, pp. 77–84, 2020.

[2] M. Herbster, "Supervised learning lecture notes series," 2020.

[3] Wikipedia, "Mercer's theorem," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Mercer%27s$_t$*heorem*

[4] S. Sharma, "Kernel trick in svm," 2020. [Online]. Available: https://medium.com/analytics-vidhya/how-to-classify-non-linear-data-to-linear-data-bb2df1a6b781

[5] N. Verma, "Perceptron and kernelization," 2021. [Online]. Available: http://www.cs.columbia.edu/ verma/classes/ml/lec/lec3$_l$*inperceptron*$_k$*ern.pdf*

[6] J. F. Trevor Hastie, Robert Tibshirani, "The element of statistical learning, 2^{nd} edition," pp. 14–18, 2008.

[7] neo4j, "The cosine similarity algorithm," 2020. [Online]. Available: https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/cosine/

[8] A. Band, "Multi-class classification — one-vs-all one-vs-one," 2020. [Online]. Available: https://towardsdatascience.com/multi-class-classification-one-vs-all-one-vs-one-94daed32a87b

[9] C. D. Sa, "The kernel trick, gram matrices, and feature extraction," 2017. [Online]. Available: https://www.cs.cornell.edu/courses/cs6787/2017fa/Lecture4.pdf

[10] V. Srikumar, "Lecture 10, the perceptron algorithm," 2018. [Online]. Available: https://www.cs.utah.edu/ zhe/pdf/lec-10-perceptron-upload.pdf