Supervised Learning (COMP0078) Assignment II

Toby Drane

January 11, 2020

Contents

1 Theoretical Background

1.1 Perceptron Algorithm

First introduced in 1958 by Frank Rosenblatt [4], the linear binary perceptron is known to be the simplest learning algorithm. We have an input space $\mathcal{X} = \mathbb{R}^d$, $\mathcal{Y} = \{-1, 1\}$. Our perceptron learning algorithm is such, that at a round t, we receive an input vector $x_t \in \mathbb{R}^d$, the algorithm maintains a weight vector $w^{(t)} \in \mathbb{R}^d$, then makes a prediction $\hat{y} = sign(\langle w^{(t)}, x_t \rangle)$. The algorithm then receives the true value $y_t \in \mathcal{Y}$, and then pays some cost to it's weights if $\hat{y} \neq y_t$ [5].

Just like most learning algorithms the Kernel trick can be applied to the input data, lifting it into higher dimensions can causing it to be linearly separable. A simple binary perceptron with kernels can be seen in algorithm 1, this can be used to predict data from within two separating classes.

Algorithm 1: Binary Perceptron Algorithm with Kernels

Data: $\{(x_1, y_1), ..., (x_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$
initialize: $w_0 = 0, (a_0 = 0)$
for $t = 1, 2, ..., T$ **do**
  receive x_t
  predict $\hat{y} = sign(\langle w_t, x_t \rangle) = sign(\sum_{i=1}^{t-1} \alpha_i K(x_i, x_t))$
  update $\alpha_t = 0, if(y_t = \hat{y}), else = y_t$
  $w_{t+1} = w_t + \alpha_t K(x_t, x_i)$
end

1.2 Support Vector Machines

A support vector machine (SVM) is a form of linear classifier, defined by a separating hyperplane. The algorithm seeks to output a optimal separating hyperplane, used to classify new, unseen data points. The first and simplest SVM algorithm is known to be the maximal margin classifier. It is important to note however, this method only works on linearly separable data.

For the following definitions we assume a binary classification problem. The training data is defined as $S = \{(x_1, y_1), ..., (x_m, y_m)\} \in \mathbb{R}^n \times \{-1, 1\}$.

1.2.1 Maximal Margin Classifier

There exists some hyperplane such that

$$h(x, w, b) = w^T x + b = \sum_{i=1}^{m} w_i x_i + b \tag{1}$$

There can exist multiple hyperplanes that all separate the training data, it is then required to chose the hyperplane that maximises the margin between the two classes, thus reducing the expected risk for a unseen data point prediction. Let's define the margin of the hyperplane γ, as the distance from the hyperplane to the closest point across both classes.

$$\gamma(w, b) = min_{x \in S} \frac{|w^T x + b|}{\|w\|_2} \tag{2}$$

Our maximal margin optimisation problem then becomes

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} w^T w \\ \text{s.t.} \quad & y_i(w^T x + b) \geq 1 \\ & i = 1, ..., m \end{aligned} \tag{3}$$

Now considering the above optimisation problem in the primal form, we can derive the dual form using Lagrangian multipliers

$$\begin{aligned} L(w,b,\alpha) &= \frac{1}{2}w^T w - \sum_{i=1}^{m}\alpha_i[y_i(w^T x_i + b) - 1] \\ \frac{\partial L}{\partial w} &= w - \sum_{i=1}^{m}\alpha_i y_i x_i = 0 \\ \frac{\partial L}{\partial b} &= -\sum_{i=1}^{m} y_i \alpha_i = 0 \\ w &= \sum_{i=1}^{m}\alpha_i y_i x_i \\ 0 &= \sum_{i=1}^{m} y_i \alpha_i \end{aligned} \tag{4}$$

Substituting the two above extreme conditions into the Lagrangian we define the dual optimisation form:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \alpha_i \geq 0, i = 1, ..., m \\ \text{and} \quad & \sum_{i=1}^{m}\alpha_i y_i = 0 \end{aligned} \tag{5}$$

Once the optimal α^* has been found using the constrained quadratic program above, the optimal variables for w^*, b^* are given by

$$\begin{aligned} w^* &= \sum_{i=1}^{m}\alpha_i^* y_i x_i \\ b^* &= y_i - w^{*T} x_i \end{aligned} \tag{6}$$

1.2.2 Soft-Margin Classifier

To account for data that is non-linearly separable and allow for misclassification of data points we are unable to use equation 5. We derive a new *soft-margin* SVM that leads to a minimal number of misclassifications by introducing a slack variable $\xi_i (i = 1, ..., m)$.

The new constraint problem then becomes

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}w^T w + C\sum_{i=1}^{m}\xi_i \\ \text{s.t.} \quad & y_i(w^T x + b) \geq 1 - \xi_i \\ \text{and} \quad & \xi_i \geq 0 \quad i = 1, ..., m \end{aligned} \tag{7}$$

Now solve this constraint problem in the same way as that of the maximal margin classifier.

$$
\begin{aligned}
L(w,b,\xi,\alpha,\beta) &= \frac{1}{2}w^T w + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(w^T x_i + b) + \xi_i - 1] - \sum_{i=1}^{m}\beta_i\xi_i \\
\frac{\partial L}{\partial w} &= w - \sum_{i=1}^{m}\alpha_i y_i x_i = 0 \\
\frac{\partial L}{\partial b} &= -\sum_{i=1}^{m}\alpha_i y_i = 0 \\
\frac{\partial L}{\partial \xi} &= C - \alpha_i - \beta_i = 0 \\
w &= \sum_{i=1}^{m}\alpha_i y_i x_i \\
0 &= \sum_{i=1}^{m}\alpha_i y_i \\
0 &\leq \alpha_i \leq C
\end{aligned}
\tag{8}
$$

Our dual quadratic program for a soft-margin SVM the becomes the below, this is the quadratic programming problem we solve within our code to create the SVM.

$$
\begin{aligned}
\max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j x_i^T x_j \\
\text{s.t.} \quad & \sum_{i=1}^{m} y_i\alpha_i = 0 \\
\text{and} \quad & 0 \leq \alpha_i \leq C \quad i = 1, ..., m
\end{aligned}
\tag{9}
$$

1.2.3 SVM Kernel Trick

We want our algorithm to have the ability to create a hyperplane in a higher dimensional feature space. Like the other algorithms we apply the kernel trick to the soft-margin SVM dual problem.

$$
\begin{aligned}
\max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j K(x_i, x_j) \\
& W(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j H_{i,j}
\end{aligned}
\tag{10}
$$

where $H_{i,j} = y_i y_j K(x_i, x_j)$. The importance to note that the Gram matrix $G_{i,j} = K(x_i, x_j)$, if this is positive definite, then $H_{i,j}$ will also be a positive definite matrix, making the problem a convex quadratic program.

1.3 Decision Trees and Random Forests

So far we have seen how linear algorithms can be used to predict between different classes, and the use of Kernels within these methods to move data to higher dimensions and making them linearly separable. These models are reliant on the quality of the Kernel. The final algorithm we will introduce within the first part of this project are known as adaptive basis function models. These dispense of the use of Kernels entirely, and more specifically the classification and regression trees (CART).

1.3.1 Classification and Regression Trees (CART)

CARTs are best known as decision trees, these attempt to recursively partition the input space into a local model for each resulting space.

$$f(x) = \mathbb{E}[y|x] = \sum_{m=1}^{M} c_m \mathbb{I}[x \in R_m] \tag{11}$$

Where c_m is the mean response, in the region m. We can correspond this to minimizing the square error in region m:

$$c_m = \frac{\sum_i^M y_i \mathbb{I}[x \in R_m]}{\sum_i^M \mathbb{I}[x_i \in R_m]} \tag{12}$$

We want the tree therefore to construct itself by deciding on what the split variables / points should be, such that it minimizes the square error in region m, solving the equation:

$$min_{R_1,\ldots,R_P} \left\{ \sum_{i=1}^{M} (y - \sum_{p=1}^{P} ave(y_j|x_j \in R_p)\mathbb{I}[x_i \in R_p])^2 \right\} \tag{13}$$

This problem is NP-Complete, as mentioned in [3], and so computationally infeasible. The approach is to use a greedy algorithm, recursive binary splitting.

Define the axis half-planes: $R_1(j,s) = \{x|x_j \leq s\}$ and $R_2(j,s) = \{x|x_j > s\}$. Now search for the optimal values of j and s:

$$min_{j,s} \left\{ min_{c1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + min_{c2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right\} \tag{14}$$

The complexity of this problem is then O(nm). Decision trees are known to be a high variance model, with little bias.

Gini Impurity The Gini impurity of a node is the probability that a randomly chosen sample in this node would be incorrectly labelled if it was labeled by the distribution of the samples in the node.

$$I_G(p) = \sum_{i=1}^{J} p_i^2 \tag{15}$$

Where p_i, is the probability of an item with label i.

1.3.2 Random Forests

Random forests were introduce to reduce this variance problem of trees. They train M different trees, on different data subsets of the input data. The ensemble is then computed $f(x) = \sum_{m=1}^{M} \frac{1}{M} f_M(x)$ [3].

What makes random forests a unique algorithm is the way they ensemble the trees from the input data. They train the M decision trees in parallel using a bagging algorithm. The trees are trained on various subsets of the dataset, using different subsets of the available features. This ensures every individual tree in the random forest is produced from unique datasets, reducing the algorithms variance.

1.4 Multiclass Linear Algorithms

We have seen some very good efficient, linear learning algorithms. However, much like everything in the real world our data is not a binary decision. We have several different digit patterns from 0-9, and so 10 different class labels. We therefore, need to find a way to generalise these learners such as the perceptron, into k-class solvers. In this project we will look at and compare two of the most common multiclass methods; the one vs all algorithm, and the one vs one algorithm.

1.4.1 One vs All Algorithm

The earliest multiclass algorithm implementation. Given k classes we construct exactly k linear prediction models. $model_i$ is trained with all data examples, $\forall i \in k$. The linear models are trained such that if i is the given class input, these are all given positive, otherwise they are all given negative. The final prediction is then the $\hat{y} = argmax_{i \in \{1,\dots,k\}} f_k(x)$.

1.4.2 One vs One Algorithm

The other common multiclass classification algorithm is the one vs one algorithm. Here we construct $k(k-1)/2$ different linear learners. Where each k learner is trained on a different pair of classes from the data. We use a similar approach to the training of the algorithm as the one used in [2]. We have our input data value (x, y), if it is in our ith class we update our weights at i by a positive increment. If the input value is in our j, likewise the jth weight gets updated by a positive increment. The prediction value $\hat{y}$, is then the index of the max classifier.

2 Algorithm Implementation and Development

Within this section we will discuss the implementations of the various algorithms used across this project. It is important to note all the algorithms were implemented using Python however, the implementations described will be at a higher, pseudo level and, therefore could be implemented in other desired languages. Throughout this project we use the widely known MNIST digit recognition dataset.

We have some input data $x \in \mathbb{R}^{m \times n}$, this is a matrix of m, n sized records. For our data we have 9298 different records (m), each one corresponding to a 16×16 grey scale image, therefore a 256 dimension data point (n), thus $X = \{-1, +1\}^{9298 \times 256}$. Along with the x our input also consists of a y vector size 9298, which is each records class for the image, e.g. a 6.

Even though the implementation of both perceptron algorithms are online, we initialise the Gram matrices. We can get away with this due to the fact we have a finite data size. This is just we don't have to compute the matrix every epoch and saving us computation time.

For both perceptron algorithms we initialise a weight matrix. Depending on what multi classification algorithm is used it will be a dimension T, by the dimension of the input data m. Where T varies on the implementation, discussed in the relevant algorithms below. This representation is like saying for every t, in the matrix, it corresponds to the weights of a single binary perceptron. One could implement this in another why by creating a single binary perceptron with its own separate logic, and depending on the algorithm, having a array of binary perceptions size T. Having both the Gram matrix and weight matrix, all the updates can then be matrix addition and multiplication. When using numpy this greatly speeds up any calculations required, and reduces the need to run different binary perceptron training methods.

2.1 One vs All Kernel Perceptron

As mentioned previously for the One vs All Perceptron we have C different binary perceptrons, where C is how many classes we wish to predict. Reflected in the weight matrix.

In our training loop we match the current weight class with the input label. If the image is 7 we are positively training the 7th binary perceptron, if they do not match with the class all other weights are

negatively valued.

Algorithm 2: 10 Class One vs All Kernel Perceptron

Data: $\{(x_1, y_1), ..., (x_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$
Classes $C = 10$
initialize Gram matrix: $G = K(X, X)$
initialize weight matrix: $W = \begin{bmatrix} 0_{1,1} & \dots & 0_{1,m} \\ \vdots & \ddots & \\ 0_{C,1} & & 0_{C,m} \end{bmatrix}$
for *until converged or epoch* $e \in Es$ **do**
  for $i = 1, 2, ..., m$ **do**
    $\hat{Y} = W \cdot G_i$
    for $c = 1, 2, ..., C$ **do**
      $y = Y_i$
      $z = \begin{cases} 1.0, & \text{if } y = c \\ -1.0, & \text{otherwise} \end{cases}$
      if $z \cdot \hat{Y}_c \leq 0$ **then**
        $W_{c,i} = W_{c,i} + z$
      end
    end
  end
end
prediction: $argmax(W \cdot K(X, x))$

2.2 One vs One Kernel Perceptron

One vs One follows the same setup but the note the difference in the size of the weight matrix. We have discussed in the previous section how each binary perceptron learns a different pair of input values. This is

the justification for the nest for statement comparing the $i, j = c$.

Algorithm 3: 10 Class One vs One Kernel Perceptron

Data: $\{(x_1, y_1), ..., (x_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$
Classes $C = 10$
initialize Gram matrix: $G = K(X, X)$
initialize weight matrix: $W = \begin{bmatrix} 0_{1,1} & \dots & 0_{1,m} \\ \vdots & \ddots & \\ 0_{C(C-1)/2,1} & & 0_{C(C-1)/2,m} \end{bmatrix}$
for *until converged or epoch* $e \in Es$ **do**
  for $d = 1, 2, ..., m$ **do**
    $\hat{Y} = W \cdot G_d$
    for $i = 1, 2, ..., C$ **do**
      for $j = i + 1, ..., C$ **do**
        $y = Y_d$
        if $y = i$ **then**
          $z = 1.0$
          if $z \cdot \hat{Y}_i \leq 0$ **then**
            $W_{i,d} = W_{i,d} + z$
          end
        end
        if $y = j$ **then**
          $z = -1.0$
          if $z \cdot \hat{Y}_i \leq 0$ **then**
            $W_{i,d} = W_{i,d} + z$
          end
        end
      end
    end
  end
end
prediction: index(list($W \cdot K(X, x)$))

For both perceptron algorithms we limited the algorithm to a maximum of 30 epochs or until the mistakes difference between the current run and the previous run was 0.01, therefore for every run the algorithm has to have an improvement of at least of 1%. We found the algorithms rarely converged before the max epoch limit was reached. The max epoch limit was found before the bulk of the experiments run. This is another trade off problem between accuracy and time complexity. We found for both algorithms 30 epochs was a good number to achieve a suitable accuracy, and not worth the time spent to run for another 10 to get a slight reduction in accuracy.

2.3 Support Vector Machine

The support vector machine implementation was created using the CVXOPT [1] package. This allowed us to solving the convex quadratic programming mentioned in the previous section where we talk about the soft-margin definition. The SVM introduced a new free parameter C, which controls the SVM trade-off between misclassifying points and thus the training, testing error. The lower the C parameter the more outliers we allow because of a larger-margin hyperplane. Conversely the larger C, the smaller the margin of the hyperplane. We can say if C tends to infinity our SVM then becomes hard-margin.

Like other free parameters we initially perform cross-validation over a different set of these parameters; $C = \{0.001, 0.01, 0.1, 1, 10, 100\}$, choosing the smallest parameter. We then decreased this set into a smaller subsize and repeat. The best value of C we found for the SVM was 0.001.

[1] `https://cvxopt.org/`

2.3.1 Trade-Offs

Due to our implementation of the SVM it was by far our slowest algorithm. In fact it was too slow to run across the whole data set, and for the 20 runs. We limited the runs to a maximum of 10 and reduced the data set by 40%. Of course this was a balancing act between accuracy and time spent running the algorithm.

2.4 Random Forest

For the random forest implementation we used the RandomForestClassifier that is included within the sklearn package. The model is trained with two dense arrays, in this case it is the m different images, with the second array corresponding to the images labels. We also have a parameter θ which is the number of decision trees which we want the ensemble model to create, this will be the parameter we shall cross-validate over to find the optimal solution.

The sklearn model follows the same implementation to the general random forest algorithm introduced by [1], as mentioned in the previous sections. The only exception to this algorithm is the classifiers are combined by an average of each probabilistic predictions, instead of an individual vote for a single class. By default it uses the Gini impurity to split the trees on and that's what we used for this project.

As discussed before random forest do not make use of Kernels methods. We therefore use the number of trees in the random forest as our parameter set to cross-validate over.

3 Part I

3.1 Polynomial Kernel

Unless stated otherwise the different experiments were all run using the polynomial kernel. Defined as the following:

$$K(x, x) = (x^T x)^d \tag{16}$$

where d is the kernel dimension, $d = \{1, 2, ..., 7\}$ throughout the project. Polynomial is a common kernel within kernelized models used for pattern analysis such as the SVM.

It is important for one to understand the impact of the kernel dimension of the algorithm. Having a lower kernel dimension such as 1, limits how much the algorithm can distinguish between the data due to the non-linear issue. On the flip side however, a higher kernel dimension causes the algorithm to over fit to the training samples.

Both of these phenomenons will become apparent throughout the different experiments.

3.2 One vs All Algorithm

3.2.1 Basic Results

For the following polynomial kernel dimension, $d = \{1, ..., 7\}$, 20 runs for performed with our One vs All perceptron algorithm. We can see the mean train, test error, as well as the standard deviations (s.d) within table 1.

k Dimension	train error & s.d	test error & s.d
1	$0.08819892 \pm \sigma' 0.01343169$	$0.18377252 \pm \sigma' 0.00489274$
2	$0.08819892 \pm \sigma' 0.00628734$	$0.00393923 \pm \sigma' 0.00135523$
3	$0.08819892 \pm \sigma' 0.00391505$	$0.00162678 \pm \sigma' 0.00048082$
4	$0.08819892 \pm \sigma' 0.00333637$	$0.00079995 \pm \sigma' 0.00050925$
5	$0.02706989 \pm \sigma' 0.00362764$	$0.00087389 \pm \sigma' 0.00052502$
6	$0.02723118 \pm \sigma' 0.00358758$	$0.00080667 \pm \sigma' 0.00043148$
7	$0.02948925 \pm \sigma' 0.00431542$	$0.00065878 \pm \sigma' 0.00046747$

Table 1: OvA train/test error mean and standard deviation

3.2.2 Cross-Validation

We want to find the optimal value of d^* for our polynomial kernel such that, it yields the best test error. We performed 20 runs, using 5-fold cross validation to find the optimal d^*. Once the best d^* was selected, the model was then retrained using this value and a test error value produced. Both of these runs were run using a 80% training data split, leaving 20% test data split.

Table 2, shows the best d^* for every run, with it's corresponding retrained test error, as well as a mean $d^* \pm$ s.d and mean test error $\pm$ s.d.

Optimal d^*	**test error**
4	0.03172
7	0.03279
4	0.03172
4	0.03172
6	0.02956
3	0.03064
5	0.03225
3	0.03064
5	0.03225
4	0.03172
3	0.03064
3	0.03064
4	0.03172
3	0.03064
5	0.03225
4	0.03172
4	0.03172
4	0.03172
4	0.03172
4	0.03172
$4.15 \pm \sigma' 1.0134$	$0.03147 \pm \sigma' 0.000750$

Table 2: OvA optimal d^* & test error

3.2.3 Confusion Matrix

A same initial setup was used as the above, that is we first find an optimal value of d^* for the polynomial kernel, repeated for 20 runs, using 5-fold cross validation. The model was then retrained again on the train, test data split using the optimal d^*. A individual 10x10 confusion matrix was produced where one was placed in cell (y, yp), where y is the true value and yp is the predicted value, such that they don't match.

The final confusion matrix was produced, where each confusion cell is the average value of the 20 individual cells at each position, with the standard deviation also noted. Table 3 displays these values, we also produced a matrix visualisation of just the confusion errors, as seen in figure 1.

		Actual Value									
		0	1	2	3	4	5	6	7	8	9
Predicted Value	0	0 ± 0	0 ± 0	0 ± 0	0.25 ± 0.433	0 ± 0	0.15 ± 0.357	0.6 ± 0.583	0 ± 0	0.8 ± 0.4	0 ± 0
	1	0 ± 0	0 ± 0	0 ± 0	0 ± 0	7.05 ± 7.123	0 ± 0	0.4 ± 0.489	0 ± 0	0.5 ± 0.5	0 ± 0
	2	0.3 ± 0.556	0 ± 0	0 ± 0	1.3 ± 0.781	2.15 ± 0.792	0 ± 0	0 ± 0	0.85 ± 0.357	0.8 ± 1.029	0 ± 0
	3	0 ± 0	0 ± 0	20 ± 0.707	0 ± 0	0 ± 0	1.1 ± 0.7	0 ± 0	0.45 ± 0.497	2.1 ± 0.994	0 ± 0
	4	0 ± 0	1 ± 0	2.1 ± 0.435	0 ± 0	0 ± 0	0 ± 0	0 ± 0	1 ± 0	0.05 ± 0.0217	0.6 ± 0.489
	5	1.05 ± 0.217	0 ± 0	0 ± 0	1.95 ± 0.668	2.7 ± 0.556	0 ± 0	1.25 ± 0.887	0 ± 0	0.5 ± 0.5	0.15 ± 0.357
	6	0.2 ± 0.4	0.1 ± 0.3	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0.05 ± 0.217	0.05 ± 0.217
	7	0 ± 0	0 ± 0	0 ± 0	0.95 ± 0.217	1.3 ± 0.781	0 ± 0	0 ± 0	0 ± 0	0 ± 0	0 ± 0
	8	0.15 ± 0.357	0 ± 0	0.8 ± 0.4	0.2 ± 0.4	1.15 ± 0.357	1.55 ± 0.497	0 ± 0	0 ± 0	0 ± 0	1.95 ± 1.071
	9	1 ± 0.707	0 ± 0	0 ± 0	0 ± 0	3.85 ± 1.061	0.75 ± 0.433	0 ± 0	1.7 ± 0.842	0 ± 0	0 ± 0

Table 3: Confusion matrix with standard deviations

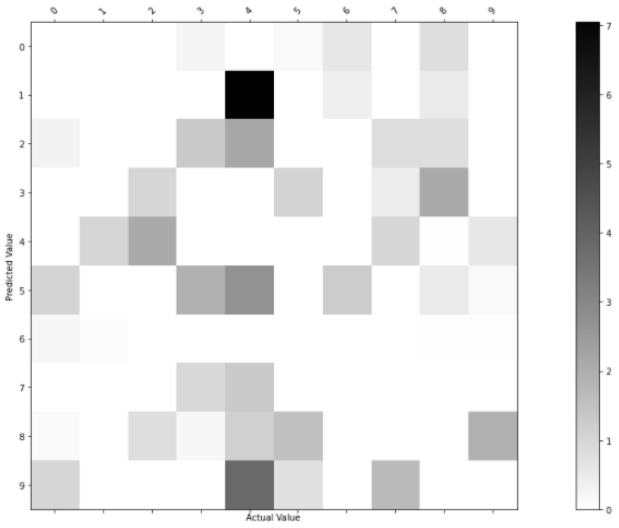

Figure 1: Confusion matrix visualisation

3.2.4 Hardest Correct Images

When we go to predict a value from within our C-class algorithm, we produce a prediction vector of size C. As mentioned above this is our confidence, $k \in \mathbb{R}^k$, where k_i is the confidence in predicting our label i. We use this algorithm confidence vector to find the five hardest to predict images, that is the for all our correctly predicted images during our test set run, what are the five smallest confidence values from this confidence vector.

When running this sub-experiment it is important that for every run the order of the data is noted. As we randomise the data when we perform our data splits, the index of the image will be different for every new data split, and therefore cannot be used consistently. This subtly was countered for by incorporating a seed value for the random data split, ensuring the indices of the images justly represented the image.

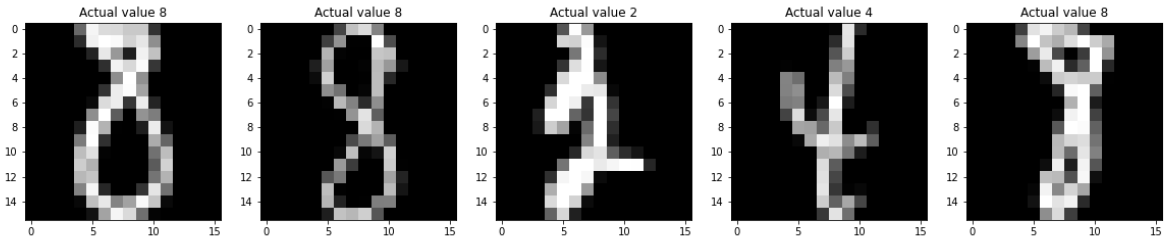

Figure 2: Five hardest to predict OvA images

We can see in figure 2, what those five hardest to correctly predict images were for our One vs All algorithm. It is no surprise to see why the algorithm struggles to correctly predict these five images. These images suffer greatly from noise adjusting their shape. The outer eights could easily be mistaken for a zero and one respectively. There is however the two tiny gaps which we can look at and symbolise with being an eight, this is probably the feature the algorithm is holding onto and correctly predicting the eight.

The two image has some bad noise, pushing it to have a similar structure to that of a three. I think if that spike at the top of the two was a little more bent to the left, the algorithm and probably us would be thinking it was more a three.

The third eight and the four are interesting ones on why the algorithm struggled to predict them. The structure is pretty good for them, albeit the four is a little "loose". They are however dimmer. They are greatly darker on their grey scale values and could be the reason for the algorithm struggling to predict them. Of course our algorithms use the $\{-1, 1\}$ values to predict the images, with darker values it could be thinking it is a blank pixel.

3.2.5 Gaussian Kernel

Within this section we now replaced the polynomial kernel, with the Gaussian Kernel, otherwise known as the Radial Basis Function (RBF).

$$K(p, q) = e^{-c\|p-q\|^2} \tag{17}$$

Here c is now our free parameter, which may also be expressed as $\frac{1}{2\sigma^2}$. To find our new parameter set for the Gaussian kernel, we initially tried a range of values $-10, 1, 0.1, 0.001, 10, 100$ and cross-validated the OvA model on each one to find the best value. We then found the best values repeated over 20 runs, and used this to drill deeper and repeating the process with a smaller subset of new dimensions. We resulted in our kernel dimension vector being $c = \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07\}$.

Basic Results We followed the same experimental setup as our first two experiments, but replacing the polynomial kernel with the Gaussian kernel as described above. 20 runs performed, noting the train and test errors along with the respective standard deviations, as seen in table 4.

c Dimension	train error & s.d	test error & s.d
0.01	$0.000806 \pm \sigma^{'} 0.000447$	$0.0263 \pm \sigma^{'} 0.00478$
0.02	$0.000309 \pm \sigma^{'} 0.000321$	$0.0274 \pm \sigma^{'} 0.00397$
0.03	$0.000242 \pm \sigma^{'} 0.000287$	$0.0298 \pm \sigma^{'} 0.00445$
0.04	$0.000114 \pm \sigma^{'} 0.000248$	$0.0342 \pm \sigma^{'} 0.00346$
0.05	$0.0 \pm \sigma^{'} 0.0$	$0.0380 \pm \sigma^{'} 0.00335$
0.06	$0.0 \pm \sigma^{'} 0.0$	$0.0419 \pm \sigma^{'} 0.00421$
0.07	$0.0 \pm \sigma^{'} 0.0$	$0.0456 \pm \sigma^{'} 0.00392$

Table 4: OvA train/test error mean and standard deviation, with Gaussian kernel.

In table 4 when we get to our largest c parameter we have a good example of the issues with the parameter for the Gaussian kernel. We know c is inversely proportional to the variance of the data within the kernel. The higher our c gets the "closer" the data gets to one another within the Kernel. This makes it harder for the algorithm to distinguish between data points causing the test error to increase.

Cross-Validation Just like for the polynomial kernel we want to find the optimal parameter for our kernel, such that after 20 runs of five-fold cross validation we have the lowest test error. It is the same experimental to that of the polynomial cross-validation setup, once the best $c*$ was selected, the model was then retrained and the test error noted. The results can be seen in table 5.

3.3 One vs One Algorithm

We use the polynomial kernel with the same parameter set $d = \{1, 2, ..., 7\}$ within our algorithm.

3.3.1 Basic Results

20 runs performed for each kernel dimension d, with the One vs One perceptron algorithm. Table 6 shows the mean train, test error of each dimension across the 20 runs, along with the relevant standard deviations.

Optimal c^*	test error
0.02	0.02903
0.01	0.03172
0.02	0.02903
0.03	0.03118
0.02	0.02903
0.02	0.02903
0.01	0.03172
0.01	0.03172
0.01	0.03172
0.02	0.02903
0.01	0.03172
0.02	0.02903
0.04	0.03602
0.02	0.02903
0.02	0.02903
0.02	0.02903
0.01	0.03172
0.01	0.03172
0.02	0.02903
0.01	0.03172
$0.0175 \pm \sigma^{'} 0.00766$	$0.03056 \pm \sigma^{'} 0.00179$

Table 5: OvA optimal c^* & test error, using Gaussian kernel

k Dimension	train error & s.d	test error & s.d
1	$0.20714106 \pm \sigma^{'} 0.01215684$	$0.14193548 \pm \sigma^{'} 0.01225641$
2	$0.00324684 \pm \sigma^{'} 0.00133722$	$0.0667473 \pm \sigma^{'} 0.0206008$
3	$0.00166039 \pm \sigma^{'} 0.00080921$	$0.05161290 \pm \sigma^{'} 0.01266012$
4	$0.0008671 \pm \sigma^{'} 0.0008072$	$0.05163978 \pm \sigma^{'} 0.01905493$
5	$0.00073945 \pm \sigma^{'} 0.00070053$	$0.04532258 \pm \sigma^{'} 0.01005405$
6	$0.00070583 \pm \sigma^{'} 0.00096377$	$0.04602151 \pm \sigma^{'} 0.00807598$
7	$0.0006318 \pm \sigma^{'} 0.0473947$	$0.04739247 \pm \sigma^{'} 0.00939928$

Table 6: OvO train/test error mean and standard deviation

3.3.2 Cross-Validation

Just like the previous cross-validations, we want to find the optimal parameter for the kernel. 20 runs were performed on a 5-fold cross validation to find the best d^*, such that it produced the lowest test error. Once the optimal kernel parameter was found the One vs One algorithm was then retrained using this d^* with the test error noted, all values can be seen in table 7.

3.4 Support Vector Machine (SVM)

We discussed previously the cross-validation value of C, that we pass as a parameter to the SVM. Along with the value for C, we use the same polynomial kernel as the other algorithms.

3.4.1 Basic Results

The experimental setup is the same as that of the previous algorithms for a basis result. 20 runs repeated for each kernel dimension with the mean train and test error, along with the respective standard deviations plotted from within table 8.

Optimal d^*	test error
4	0.043010
5	0.034408
6	0.040860
6	0.040860
6	0.040860
4	0.040860
5	0.034408
6	0.040860
5	0.034408
4	0.043010
5	0.034408
4	0.043010
6	0.040860
5	0.034408
5	0.034408
5	0.034408
7	0.039021
4	0.043010
5	0.034408
7	0.036021
$5.2 \pm \sigma^{'} 0.92736$	$0.03833 \pm \sigma^{'} 0.003709$

Table 7: OvO optimal d^* & retrained test error

k Dimension	train error & s.d	test error & s.d
1	$0.050544 \pm \sigma^{'} 0.0$	$0.02066 \pm \sigma^{'} 0.0271$
2	$0.050883 \pm \sigma^{'} 0.0135$	$0.02080 \pm \sigma^{'} 0.0542$
3	$0.050826 \pm \sigma^{'} 0.0677$	$0.02071 \pm \sigma^{'} 0.0271$
4	$0.05068 \pm \sigma^{'} 0.0672$	$0.02071 \pm \sigma^{'} 0.0281$
5	$0.05067 \pm \sigma^{'} 0.0$	$0.02075 \pm \sigma^{'} 0.0543$
6	$0.054074 \pm \sigma^{'} 0.0$	$0.02207 \pm \sigma^{'} 0.0212$
7	$0.05977 \pm \sigma^{'} 0.0488$	$0.02442 \pm \sigma^{'} 0.0495$

Table 8: SVM train/test error mean and standard deviation

3.4.2 Cross-Validation

We want to find the optimal polynomial dimension for our SVM, so 20 runs of 5 fold cross validation were performed for all values of d, to find the d^*. Once they were found the SVM was retrained on this d^* and the test error for the new retrained SVM found. Table 9 shows these results.

3.5 Random Forest

Before we stated the random forest is an adaptive basis function algorithm such that it does not require the kernels data, for the random forest experiments instead of trying to find an optimal value of d for the kernel dimension, we wanted to find the optimal value of m, that corresponds to how many classifiers to build inside the random forest.

3.5.1 Basic Results

The first basic run was 20 repeats for every value of our m, resulting in the mean train and test errors, seen in table 10.

Optimal d^*	test error
4	0.01633
4	0.01633
5	0.01731
4	0.01633
5	0.01731
5	0.01731
4	0.01633
5	0.01731
5	0.01731
5	0.01731
5	0.01731
5	0.01731
4	0.01633
4	0.01633
5	0.01731
1	0.01644
4	0.01633
4	0.01633
1	0.01644
3	0.01641
$4.1 \pm \sigma^{'} 1.17898$	$0.016789 \pm \sigma^{'} 0.047586$

Table 9: SVM optimal d^* and retrain test error

m Dimension	train error & s.d	test error & s.d
20	$0.00010083 \pm \sigma^{'} 0.00009387$	$0.047069 \pm \sigma^{'} 0.005149$
30	$0.00005377 \pm \sigma^{'} 0.00007839$	$0.042768 \pm \sigma^{'} 0.002956$
40	$0.00001344 \pm \sigma^{'} 0.00004033$	$0.040510 \pm \sigma^{'} 0.005126$
50	$0.00000672 \pm \sigma^{'} 0.00002930$	$0.039784 \pm \sigma^{'} 0.003724$
60	$0.00000672 \pm \sigma^{'} 0.00002930$	$0.036881 \pm \sigma^{'} 0.003523$
70	$0.0 \pm \sigma^{'} 0.0$	$0.036774 \pm \sigma^{'} 0.004854$
80	$0.0 \pm \sigma^{'} 0.0$	$0.037715 \pm \sigma^{'} 0.003918$

Table 10: RF m dimension, mean train and test errors with SD

3.5.2 Cross-Validation

Likewise for the kernel dimension the 5-fold cross validation was used to find the optimal value of m for the random forest, this is what is the best number of classifiers to produce the ensemble upon, results in table 11.

4 Comparison

We have run four different algorithms, three with kernel data and one without the use of Kernels. In table 12 we draw a comparison between all of the algorithms. We use the cross-validation score as the means for the comparison.

We can see all five of the models have a very similar error, getting about 97% accuracy across the test set. Our worst model, but only very slightly worse is the random forest. Being an adaptive basis function model might be hindering the model to distinguish correctly between the different data items, and the potential over fitting of the training sets. It is however only very slightly worse therefore, cannot be discounted as a use case for any future experiments. The best model was the SVM. This was surprising initially because of the fact we trained our SVM on a slightly smaller dataset due to the computation time it took for the

Optimal m^*	**test error**
40	0.03709
80	0.03763
80	0.03763
70	0.03763
70	0.04139
50	0.03655
60	0.04408
80	0.03602
80	0.03763
70	0.03763
70	0.03924
80	0.03978
80	0.03709
70	0.03763
60	0.03978
80	0.03817
80	0.04032
70	0.04247
80	0.03655
60	0.03817
$70.5 \pm \sigma^{'} 11.16915$	$0.03862 \pm \sigma^{'} 0.002056$

Table 11: RF optimal m^* & retrained test error

Algorithm	**test error**
OvA Polynomial Kernel	$0.03147 \pm \sigma^{'} 0.000750$
OvA Gaussian Kernel	$0.03056 \pm \sigma^{'} 0.00179$
OVO	$0.03833 \pm \sigma^{'} 0.003709$
SVM	$0.016789 \pm \sigma^{'} 0.047586$
Random Forest	$0.03862 \pm \sigma^{'} 0.002056$

Table 12: Algorithms Comparison

model to run. What this shows is the power the SVM has, and why it is fast becoming the most popular supervised learning model.

We saw not a huge difference between the time it took for both perceptron models to run. In the design there is a difference in the OvO model training many more classifiers than the OvA, however each classifier is trained on a smaller dataset. The time difference was very small between them, OvO only taking slightly longer.

We noticed a time difference between the polynomial and Gaussian kernel. The Gaussian kernel was alot more computationally demanding than the polynomial kernel, due to the exponential calculation.

References

[1] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[2] Chih-Wei Hsu and Chih-Jen Lin. "A comparison of methods for multiclass support vector machines". In: *IEEE Transactions on Neural Networks* 13.2 (2002), pp. 415–425. DOI: 10.1109/72.991427.

[3] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[4] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[5] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.