Supervised Learning Coursework 2

Team : 21093512, 18091452

May 10, 2022

1 Part 1

In the following report we summarise some experiments conducted with the Kernel Perceptron algorithm on a reduced version of the popular MNIST dataset (`zipcombo.dat`). In its simplest form, the perceptron algorithm is an online binary classifier. However, in the case at hand, the perceptron was adapted to be used with kernels, and generalised to a multi-class setting. In the first set of experiments (1-5), a "one vs all" approach to train the multi-class kernel perceptron was adopted. The implementation can be summarised as follows:

Algorithm 1 "One vs All" Multi-Class Kernel Perceptron (online)

```
Input: $\{(\boldsymbol{x_1},\boldsymbol{y_1}),\dots,(\boldsymbol{x_i},\boldsymbol{y_i}),\dots,(\boldsymbol{x_m},\boldsymbol{y_m})\}$ where $\boldsymbol{x}\in\mathbb{R}^n$, $\boldsymbol{y}\in\{-1,+1\}^k$ , $k\in\mathbb{N}$
  where with $\boldsymbol{y}\in\{-1,+1\}^k$ we indicate that $y_{i,j}=+1$ if $j$ = label, and -1 otherwise (an indicator function).
Define: $\mathbf{K} = \begin{bmatrix} k(\boldsymbol{x_1},\boldsymbol{x_2}) & k(\boldsymbol{x_1},\boldsymbol{x_2}) & \dots & k(\boldsymbol{x_1},\boldsymbol{x_m}) \\ k(\boldsymbol{x_2},\boldsymbol{x_1}) & k(\boldsymbol{x_2},\boldsymbol{x_2}) & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ k(\boldsymbol{x_m},\boldsymbol{x_1}) & \dots & \dots & k(\boldsymbol{x_m},\boldsymbol{x_m}) \end{bmatrix}$    ▷ This is what we'll refer to as the Kernel matrix
Initialise: $\boldsymbol{\alpha}=\mathbf{0}$, $\boldsymbol{\alpha}\in\mathbb{R}^{k\times m}$    ▷ Array of shape (k_classes, m_samples)

Training:
  for $t$ in range(m_samples) do
    $\boldsymbol{w_t} = \boldsymbol{\alpha}$[:,:t] · $\mathbf{K}$[:,:t]    ▷ Shown in array indexing notation. Equivalent to Sum 1
    for $i$ in k_classes do
      if $\boldsymbol{y_{t,i}} \odot \boldsymbol{w_{t,i}} \leq 0$ then
        $\boldsymbol{\alpha_{t,i}} = \boldsymbol{\alpha_{t,i}} -$ sign($\boldsymbol{w_{t,i}}$)
      end if
    end for
  end for

Prediction:
Initialise: $\hat{\boldsymbol{Y}} = \mathbf{-1}$, $\hat{\boldsymbol{Y}}\in\{-1,+1\}^{m\times k}$    ▷ Initialised as matrix of shape (m_samples, k_classes)
  $\boldsymbol{W} = (\boldsymbol{\alpha}\cdot\boldsymbol{K})^T$    ▷ This will also have shape (m_samples, k_classes)
  for i in $m$ rows do
    $\hat{\mathbf{Y}}_{i,j} = \arg\max_{j\in k} \boldsymbol{W}_i$
  end for
```

A central part of the training and updating procedure is the evaluation of the sum:

$$\boldsymbol{w}(\cdot) = \sum_{i=0}^{m} \alpha_i K(\boldsymbol{x_i}, \cdot) \tag{1}$$

with which the predictions $\boldsymbol{w}$ are updated after each sample.

Given that we are working with a medium to large dataset, it is essential that this sum is computed efficiently every time we "see" a new data point during online training. Here is a breakdown of how this was achieved, and how it was already reported in Algorithm 1:

- α was initialised as an empty array of shape `(n_classes, m_samples)` and updated after "seeing" each data point according to algorithm 1.
- The full kernel matrix was computed prior to training, and thus only accessed from memory during the training process. This had shape `K_train=(m_train_samples, m_train_samples)`, or, in the case of testing, `K_test=(m_train_samples, m_test_samples)`.

- At each time step `t`, the following matrix multiplication was computed (shown in python array indexing notation): `W = alpha[:,:t] @ K_train[:t,t]`. The result of this operation is the prediction vector for the sample at `t`, and has shape `W = (n_classes, 1)`. From the properties of matrix multiplication and the shape of the kernel matrix, it directly follows that this implementation is equivalent to Sum(1).

While this implementation is conceivably not the fastest nor most memory-efficient way to compute Sum (1), it has the advantage of incorporating evaluation and addition of new terms in one step by exploiting properties of matrix multiplication and the structure of the kernel matrix. Moreover, evaluation on the test set is conveniently reduced to a simple matrix multiplication which can be efficiently computed in one step.

As far as training epochs are concerned, the principle just described to compute 1 is still directly applicable if we define alpha and the kernel matrix to respectively have shapes `(n_classes, m_samples * n_epochs)` and `(m_train_samples * n_epochs, m_train_samples * n_epochs)`, or, analogously, for the testing kernel matrix `K_train=(m_train_samples * n_epochs, m_test_samples)`. This is achieved by simply tiling the computed kernel matrices for the required number of epochs.

Having summarised how some of the main points of the algorithm were implemented, let us now turn to a discussion of the different experimental protocols.

1.1 Protocol 1

In protocol 1, 20 runs of different random train/test splits of the kernel perceptron were performed. In each run, a polynomial kernel of the form $K(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i} \cdot \boldsymbol{x_j})^d$ was used, with $d = \{1, 2, ..., 7\}$. The following table summarises training and testing errors for different polynomial orders, averaged across 20 runs plus or minus standard deviation.

d	**Train error**	**Test error**
1	$7.88\% \pm 1.37\%$	$9.75\% \pm 1.49\%$
2	$1.14\% \pm 0.52\%$	$4.26\% \pm 0.67\%$
3	$0.33\% \pm 0.17\%$	$3.40\% \pm 0.41\%$
4	$0.20\% \pm 0.13\%$	$3.05\% \pm 0.34\%$
5	$0.15\% \pm 0.41\%$	$2.99\% \pm 0.51\%$
6	$0.11\% \pm 0.29\%$	$3.01\% \pm 0.48\%$
7	$0.11\% \pm 0.31\%$	$3.02\% \pm 0.47\%$

Table 1: Train and test errors of the "one vs all" multi-class kernel perceptron algorithm for 20 runs on different 80% train, 20% test splits.

It is important to note, in this case, that the kernel perceptron was always run for a fixed number of 5 epochs, independent of the degree used in the polynomial kernel. This was motivated by the fact that higher polynomial degrees (i.e. 6, 7) converge to a train error ≈ 0 after 5 epochs. Even though the use of adaptive epochs (i.e., different for each polynomial degree) was considered, this was not chosen for a number of different reasons. First of all, convergence is very unlikely to be reached in our linearly–inseparable case with low polynomial degrees (e.g. 1, 2), meaning that an arbitrary cut-off for the epochs needed to be chosen anyway for those cases. Secondly, given that we are operating in a time-constrained setting, fixing the number of epochs is a way of assessing also "speed of convergence" which is a desirable property of the algorithm. Lastly, and perhaps more practically, given our implementation of the epochs each new epoch takes more memory and time to compute. Fixing the epochs was a simple way to control and predict runtimes and resources.

1.2 Procotol 2

In protocol 2, 20 runs of 5-fold cross validation on different 80% random splits of the training set were performed to select the best values of the parameter d (representing the degree of the polynomial kernel). After computing the "best" values d^* of the parameter d with cross validation, a run on the 20% test set was performed with that value. Results of these 20 runs are summarised in the following table:

d^*	**Test error**
5.85 ± 0.91	$3.16\% \pm 0.51\%$

Table 2: Mean d^* and mean testing error plus or minus standard deviation over 20 runs of cross validation.

Note that even in this case the number of epochs for each run of the kernel perceptron was fixed to 5, both during cross validation and testing, for the reasons detailed above.

1.3 Protocol 3

During calculation of the test set errors in protocol 2, each mistake was recorded, together with its true label. This allowed for the computation of a *confusion matrix* for each of the test runs in protocol 2. Figure (X) shows the mean confusion matrix of the 20 runs, together with its standard deviation matrix. Note that here we plotted the rate of confusion, i.e. number of errors on a certain item divided by the total number of errors made on that run. Therefore, each entry of the confusion matrix represents the percentage contribution towards the total test error of a particular `(predicted, true_value)` pair.

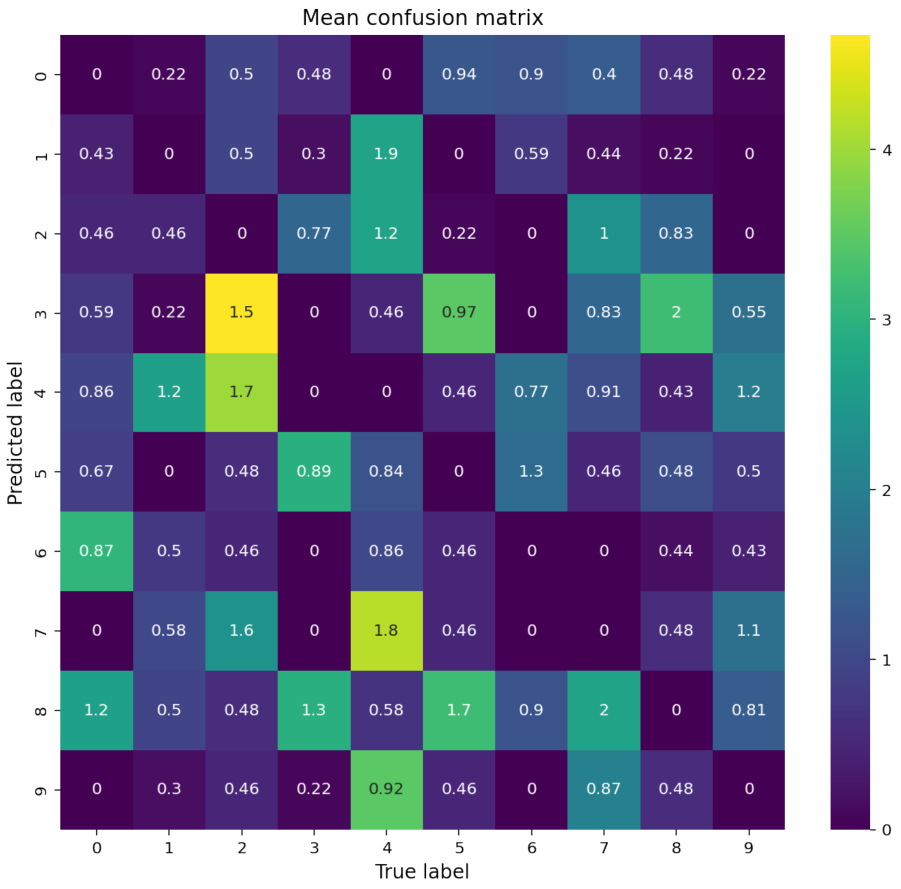

Figure 1: Mean confusion matrix and standard deviation (as numbers printed inside each cell) over 20 test runs of the Polynomial kernel perceptron. The colour of each entry of the matrix represents the confusion rate (and contribution towards total test error) of that particular pair.

It is perhaps interesting to note how certain digits are almost never confused (e.g. 0s for 1s), and how some mistakes we might regard as "human" are also made relatively often (e.g. 3s for 8s).
Interpreting the confusion matrix can give us insights into weaknesses of our model, and consequently inform us towards what types of new data we might want to include to improve performance.

1.4 Protocol 4

Another way to diagnose our model and to make sure it behaves as expected is to directly visualise digits that are incorrectly classified. This was done in protocol 4. Each misclassification in the test set of protocol 2 was recorded, along with its correct label, allowing the construction of a list of most misclassified images. Given that we performed 20 runs with 20% of the data randomly represented in the test set at each run, it is reasonable to assume we have near-balanced presentation of each image across the 20 runs. Therefore, these images would represent the hardest to predict under our model. A visualisation of the five hardest to predict images along with their correct labels is shown in Figure 2.

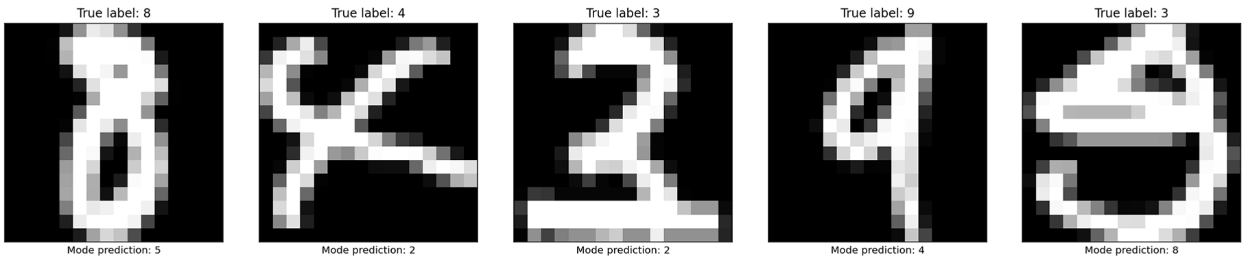

Figure 2: The five hardest to predict images during the experiments in Protocol 2. Each image is shown with its true label and the model's modal (most common) wrong prediction. The images shown here were wrongly classified respectively, from left to right, 14, 12, 11, 11 and 11 out of 20 times.

It is not that surprising that these images are hard to predict. Some of them definitely challenge even a human observer, while for others it is clear how they could be misclassified (for instance by being very similar to another digit, or being very dissimilar from the stereotypical shape of their true class). In Figure 3, a broader selection of difficult to classify images is shown, for purely illustrative purposes. Even by looking at this larger sample, the behaviour of our model remains sensible.

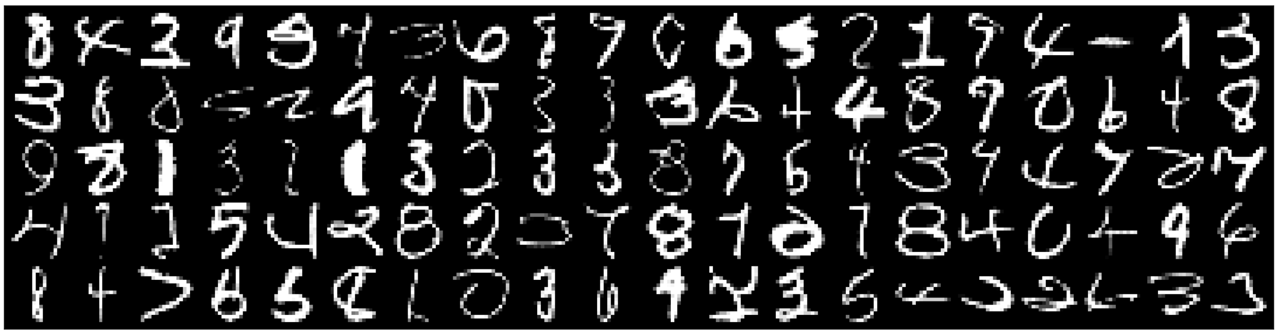

Figure 3: Larger sample of the 100 images which are the hardest to predict for the polynomial kernel perceptron. Some of them are definitely baffling even for a human.

1.5 Protocol 5

In protocol 5 we repeat the experiments conducted in 1.1, 1.2, with a Gaussian Kernel of the form:

$$K(\boldsymbol{x_i}, \boldsymbol{x_j}) = \exp\left(-\gamma ||\boldsymbol{x_i} - \boldsymbol{x_j}||^2\right) , \quad \text{with:} \quad \gamma = \frac{1}{2\sigma^2} \geq 0$$

This is described by the free parameter γ, for which a good range of values to cross-validate over was established in a “binary search” like fashion. An initial, broad, range of values was chosen as $S = [0, 2]$, and seven evenly-spaced values selected in this interval. A full run of the Gaussian kernel perceptron on a 80% train, 20% test split was then carried out, recording test losses for each γ value in the interval. The interval was then sequentially narrowed down through binary search by keeping the bound with the lower test loss and halving the bound with the higher. The final range of values chosen to conduct cross validation, after 10 steps of binary search, was:

$$S = \{0.03125, 0.0345, 0.03775, 0.041, 0.04425, 0.0475, 0.05075\}$$

Note that for all experiments in protocol 5, the same "fixed epochs" criterion as in previous sections was adopted. However, even though the Gaussian kernel perceptron almost always converged after only 3-4 epochs, it was nonetheless run for 5 epochs in all experiments in order to have a more direct and interpretable comparison with the performance of the Polynomial kernel implementation.
It is possible that this choice might have led to overfitting, and that early stopping would have let us achieve lower test errors. This will be discussed in more detail later.

For now, Table 3 summarises the equivalent experiment to 1.1 performed with the Gaussian kernel Perceptron for different values of S:

γ **value**	**Train error**	**Test error**
0.03125	$0.02\% \pm 0.01\%$	$3.05\% \pm 0.35\%$
0.0345	$0.02\% \pm 0.01\%$	$3.19\% \pm 0.37\%$
0.03775	$0.03\% \pm 0.02\%$	$3.29\% \pm 0.33\%$
0.041	$0.02\% \pm 0.01\%$	$3.37\% \pm 0.51\%$
0.04425	$0.04\% \pm 0.04\%$	$3.76\% \pm 0.51\%$
0.0475	$0.02\% \pm 0.03\%$	$3.80\% \pm 0.47\%$
0.05075	$0.02\% \pm 0.01\%$	$3.90\% \pm 0.45\%$

Table 3: Train and test errors of the "one vs all" multi-class, Gaussian kernel perceptron algorithm for 20 runs on different 80% train, 20% test splits.

Table 4 in turn summarises the equivalent experiment to 1.2 performed with the Gaussian kernel perceptron.

γ^*	**Test error**
0.0322 ± 0.0015	$3.21\% \pm 0.17\%$

Table 4: Mean γ^* and mean testing error plus or minus standard deviation over 20 runs of cross validation.

It might be interesting at this point to draw some comparisons between the performance of the Gaussian and the Polynomial kernel perceptron.
While for some choice of hyperparameters both models seem to converge to ≈ 0 training error, the Gaussian version does so more reliably, as we can see by inspecting the standard deviations in the equivalent Protocol 1 experiments. This behaviour might be expected given the properties of the Gaussian kernel, which can be interpreted as fitting a radial basis function for each sample m. For this reason, the dimensionality of the feature map associated with the Gaussian kernel will always be higher than the polynomial, and thus also its capacity to fit complex classification boundaries arbitrarily well. Even though this behaviour is often desirable, it can in some instances lead to overfitting.

In the case at hand, for example, by examining the Protocol 2 results for both kernels we can see that the test error is surprisingly close for both implementations, despite the Gaussian kernel version showing faster convergence in the training set and lower and more stable training error. Once again, however, the standard deviation for the results of the Gaussian kernel is lower than the polynomial, indicating comparable bias for the models but lower variance for the Gaussian implementation.

As it was already mentioned, it was decided to run both versions of the algorithm for a fixed number of 5 epochs, despite the Gaussian kernel perceptron converging after 3-4 epochs on average. This has some interesting implications. First of all, in time constrained settings, the Gaussian kernel perceptron should be preferred over the polynomial implementation. Secondly, it is conceivable that this decision led to an overfitting effect. Early stopping of the Gaussian kernel perceptron before complete learning of the training set is likely to reduce the bias on the test set, and make this implementation an even more preferable choice.

All things considered, given that both implementations of the algorithm had comparable runtimes and results, the Gaussian kernel perceptron should nonetheless be preferred for its lower variance, faster convergence and possibility of reducing the bias on the test set by early stopping.

1.6 Protocol 6

In protocol 6 we describe a different method to generalise the kernel perceptron to k-classes.

Up to this point, all experiments used a "one vs all" approach, where we train 10 different classifiers that learn to differentiate a given class from all others, and then predict with the most confident classifier. In this section we will use another popular technique to generalise binary classifier to k-classes: "one vs one". Using this approach, we learn $c = k(k-1)/2$ binary classifiers, where k is the number of classes we want to generalise to. In our case, this amounts learning 45 different classifiers, each distinguishing one digit from another. Predictions on a new instance are then made by means of a majority vote among the c classifiers.
The detailed implementation can be accessed in the supplementary code `Part_1_1v1.ipynb`.

The results of running this alternative generalisation to k classes are summarised in the following tables.

d	**Train error**	**Test error**
1	$4.33\% \pm 0.80\%$	$7.30\% \pm 1.01\%$
2	$0.66\% \pm 0.21\%$	$3.82\% \pm 0.43\%$
3	$0.26\% \pm 0.23\%$	$3.51\% \pm 0.42\%$
4	$0.11\% \pm 0.07\%$	$3.37\% \pm 0.39\%$
5	$0.06\% \pm 0.04\%$	$3.27\% \pm 0.50\%$
6	$0.05\% \pm 0.04\%$	$3.35\% \pm 0.27\%$
7	$0.03\% \pm 0.02\%$	$3.50\% \pm 0.35\%$

Table 5: Train and test errors of the "one vs one" multi-class polynomial kernel perceptron algorithm for 20 runs on different 80% train, 20% test splits.

d*	**Test error**
5.4 ± 1.02	$3.28\% \pm 0.31\%$

Table 6: Mean d^* and mean testing error plus or minus standard deviation over 20 runs of cross validation for the "one vs one" implementation of the polynomial kernel perceptron.

Note that, to facilitate interpretation and comparisons with previous experiments, the number of epochs was fixed to 5 even in this implementation. It can be observed that the results of the "one vs one" implementation do not diverge significantly from the "one vs all" generalisation to k classes. While there is a marginal improvement in performance on the training set for all polynomial degrees, performance on the test set remains essentially the same. Both our generalisation to k classes strategies had comparable runtimes, but given that the "one vs all" implementation required more pre-processing of the data to generate 45 "one vs one" datasets to be used in binary classification, the "one vs all" strategy might be preferred in time- and memory-constrained settings, especially if dealing with large datasets.

2 Part 2

2.1 Experiments

2.1.1 Implementation of Spectral Clustering

The spectral clustering algorithm is implemented in three experimental settings. For all three problems, the algorithm is implemented using the same procedure. The data is assumed to consist of n points, $x_1, ..., x_n$ that belong to classes $y_n := \{-1, 1\}$. One can imagine some notion of non zero similarity between these points that resembles a distance metric. In this case, the goal of clustering would be to cluster points that are similar and differentiate ones that are dissimilar. To do so, we consider the dataset to be a graph. This graph is fully connected, where each vertex represent a data point and each edge represent this similarity metric, mentioned above. In order to implement this notion of similarity, each edge of the graph is weighted. This weight is computed using the adjacency matrix, $W_{i,j} =^{-c*||x_i-x_j||^2}$. This matrix measures the distance between points given a scale parameter, c. Once W is computed, we compute the graph Laplacian matrix, L and derive its eigensystem. We the let v_2 denote the eigenvector with second smallest eigenvalue. The sign of v_2 will determine the class of every data point.

2.1.2 Experimental Results

When clustering the `twomoons.dat` data set, the optimal c value was $\mathbf{c} = \mathbf{21.11}$, as the algorithm clustered all points correctly (as shown in figure 4).

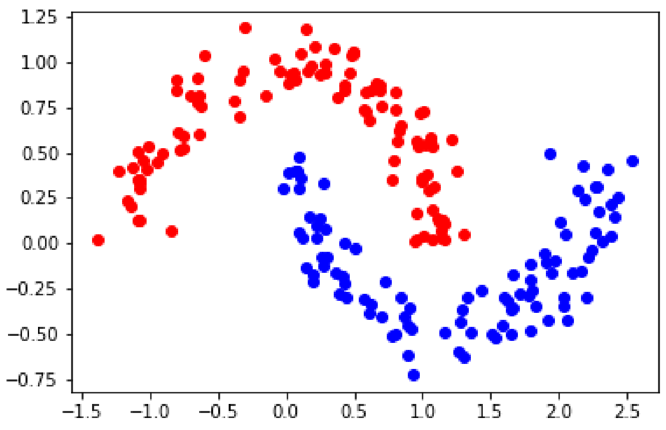

Figure 4: Correct clustering of the `twomoons` dataset

When clustering the `gaussian-clusters` data set, the optimal c value was $\mathbf{c = 1.74}$, as the algorithm clustered all points correctly (as shown in figure 5).

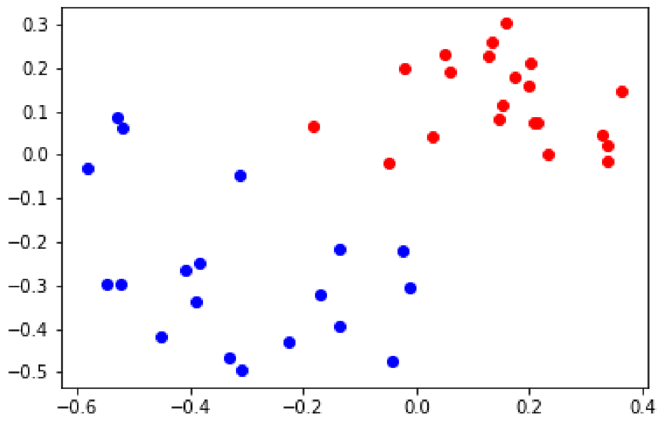

Figure 5: Correct clustering of the `gaussian-clusters` dataset

Finally, when clustering the `dtrain123` digit data set, the optimal c value was $\mathbf{c = 0.0412}$, as the misclassification was at a minimum of 0.068. As the data is high dimensional, the clustering can not be visualised directly (i.e. without using dimensionality reduction techniques).

The correct cluster percentage is another performance evaluation metric used for clustering algorithms:

$$CP(\sigma) = \frac{max(l_+, l_-)}{l}, \tag{2}$$

where l_+ is the amount of correctly classified data points, l_- is the amount of incorrectly classified data points and l is the total amount of predictions.

When using this metric to measure the quality of spectral clustering, we obtain an optimal clustering for a σ value of **0.044** for a clustered percentage of $CP(\sigma = 0.044) = 0.93157$ (see figure 6). Figure 6 illustrates the relationship between correctness and σ values for the above dataset (`dtrain123`).

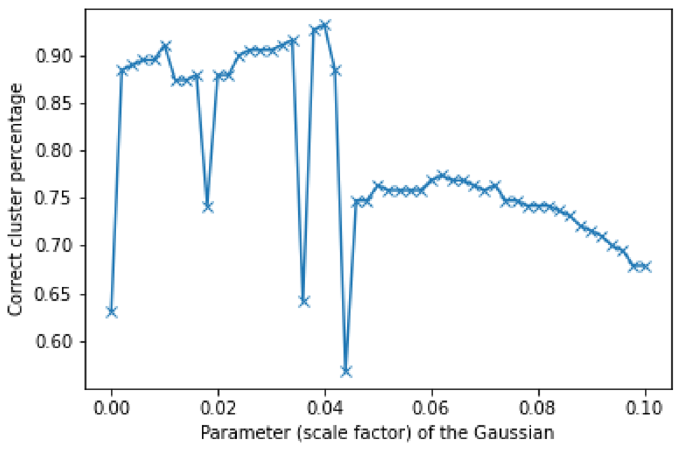

Figure 6: Plot of the relationship between cluster correctness and scale parameter σ

2.2 Questions

2.2.1 Evaluation of CP(c) as a reasonable measure of cluster correctness

As described in section 2.1.2, the correct cluster percentage is a performance evaluation metric used for clustering algorithms. It takes the maximum between the amount of correctly and incorrectly classified points and divides that quantity by the total amount of classifications. Given that correct and incorrect are only 2 possible outcomes of the algorithm, the CP metric is bounded above by 0.5 as one could inverse the prediction if $CP(\sigma) < 0.5$.

It can deemed a reasonable measure of correctness by virtue that is a misclassification error (ratio of correctly classifies/all classifications) bounded by 0.5.

2.2.2 First eigenvalue and eigenvector of the Laplacian

Let us explain why the first eigenvalue of the Laplacian is zero and its corresponding eigenvector is the constant vector. To do so we will show that

$$f'Lf = \frac{1}{2}\sum_{i,j=1}^{n} W_{i,j}(v_i - v_j)^2,$$

holds for any vector v.

$$L = D - W \equiv f'Lf = f'Df - f'Wf \tag{3}$$

$$= \sum_{i=1}^{n} d_i v_i^2 - \sum_{i,j=1}^{n} v_i v_j w_{ij} \tag{4}$$

$$= \frac{1}{2}\left(\sum_{i=1}^{n} d_i v_i^2 - 2\sum_{i,j=1}^{n} v_i v_j w_{ij} + \sum_{j=1}^{n} d_j v_j^2\right) \tag{5}$$

$$= \frac{1}{2}\sum_{i,j=1}^{n} W_{i,j}(v_i - v_j)^2 \tag{6}$$

Assuming that v is an eigenvector with eigenvalue 0. Then,

$$0 = f'Lf = \sum_{i,j=1}^{n} W_{i,j}(v_i - v_j)^2$$

Given that weights, w_{ij} represent scaled distances, they must be positive. Hence, the quantity $(v_i - v_j)^2$ must vanish for a zero eigenvalue. From this, it follows that v_i and v_j must be equal, and that v must be constant for all connected vertices. Moreover, as all vertices of a connected component in an undirected graph can be connected by a path, v needs to be constant on the whole connected component. Therefore, the eigenvalue is the constant one vector $\mathbf{1}$ with eigenvalue 0, which, trivially, is the indicator vector of the connected component.

Reference used: 2.1.1

2.2.3 Why does Spectral Clustering work?

Spectral clustering is used to cluster data using similarity graphs. Part 2.1.1 goes over how the clustering problem can be represented using a similarity graph, and how this is derived.

Reformulating the problem statement, we can say that we want to find a partition of the similarity graph such that the edges between different groups have a very low weight (which means that points in different clusters are dissimilar from each other) and the edges within a group have high weight (which means that points within the same cluster are similar to each other). This situation can be visualised as a graph cut. The graph cut will separate the data given this described partition and therefore create clusters.

Reference used: [1].

We can also used the idea of effective resistance to explain why the algorithm works. For a pair of vertices of a graph, the effective resistance of the total network when a voltage source is connected across them. In term of representing clusters, this metric takes into account connectivity and geodesic distance, unlike a classic distance metric (euclidean) that only considers geodesic distance.

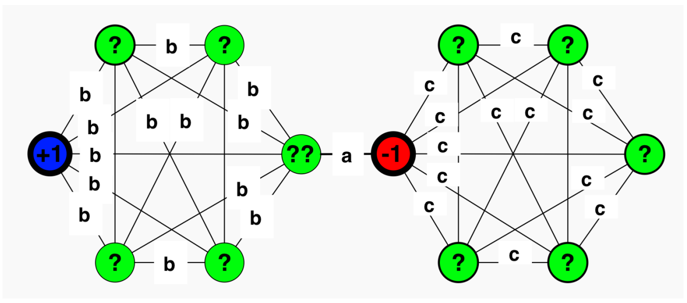

Figure 7: Using effective resistance to explain spectral clustering

The base global "distance" $d(p, q)$ is adapted according to the empirical distances via the effective resistance $r(p, q)$. Consider two cliques connected by an edge with distances $a < b$, as shown in figure 7. The effective resistance between vertices within a clique (cluster) whose edge-resistance are $\leq d$ where $r_{clique} \leq \frac{2d}{m}$. In order to cluster the points we can say that if $2b < a$ then label "+1".

2.2.4 How does the scale parameter affect the clustering?

The weight matrix W measures the distance between points given a scale parameter, c. This scale parameter is crucial as it helps minimise the distance for near points and maximise it for further away points. An optimal c value generates a clustering with minimum error (ie. where the minimum amount of clustering mistakes are made). This optimal c will depend on the dataset that we are clustering, as the distances between the points are not guaranteed to be on the same scale (see section 2.1.2).

3 Part 3

In this part, four classification algorithms are implemented and used to estimate their sample complexity. We are examining the perceptron, winnow, least-squares and $1 - NN$ algorithms.

a.

The following plots illustrate the sample complexity of all 4 algorithms.

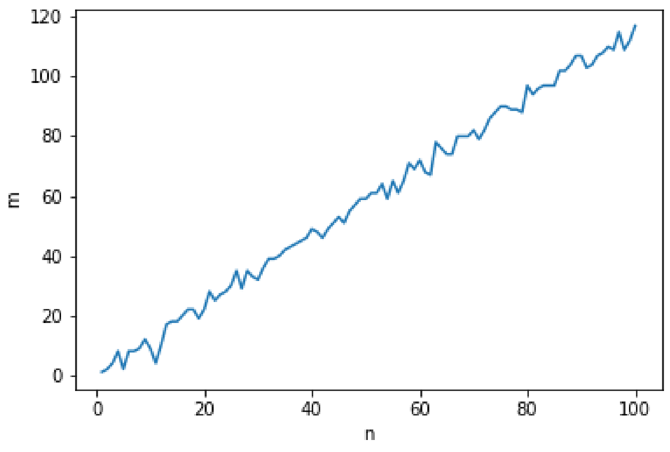

Figure 8: Sample complexity of the perceptron algorithm

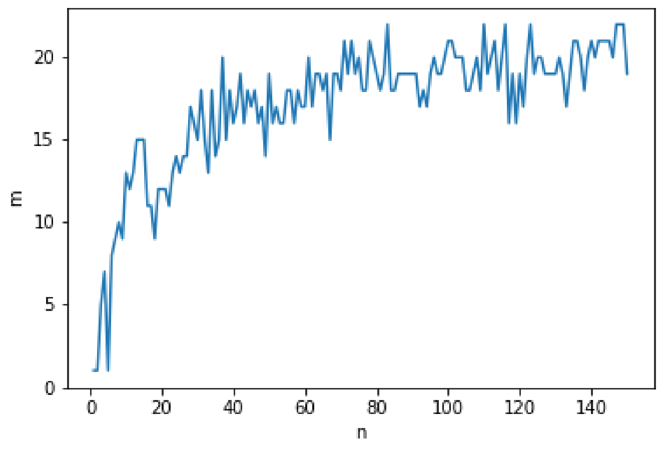

Figure 9: Sample complexity of the winnow algorithm

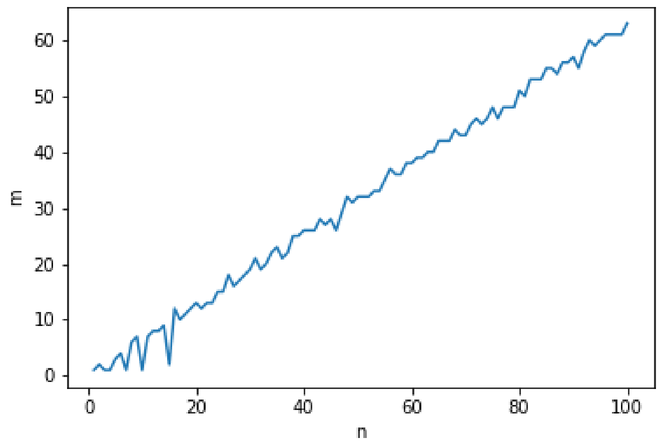

Figure 10: Sample complexity of the least squares classification algorithm

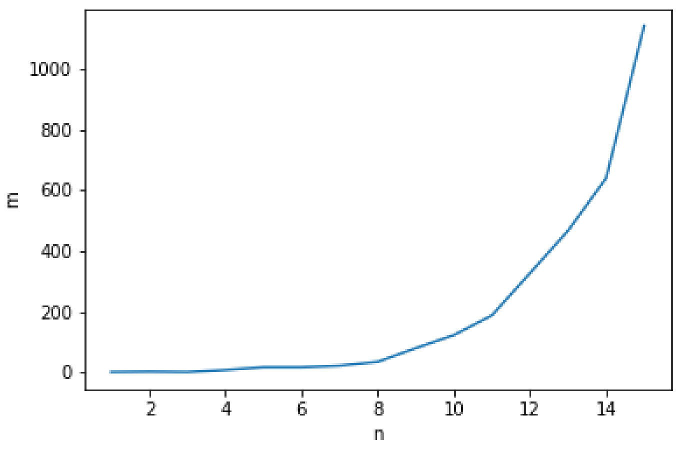

Figure 11: Sample complexity of the $1 - NN$ classification algorithm

b.

We are modelling a situation where we have training data of length m and dimension n. In order to obtain the sample complexity, we are computing the optimal m for every n. This optimal m corresponds to the training set size that allows the model to guarantee an average generalisation error of less than 0.1.

The algorithm looks for an optimal m value for every possible dimension of the data (iteration of step 1). At each iteration the optimal m is computed. To do so, we start with a value of $m = 1$. The training and testing datasets are randomly generated, and the algorithm is run. If the error requirement is satisfied, the algorithm stops. If not, we move on to $m = 2$. This process is run until an optimal value is found. To guarantee this convergence, the process is run on a `while True` condition, and will only break when the optimal value is found.

The choice of how to compute the error is crucial for an accurate estimate of the sample complexity.

Initially, we decided to compute the error using a Monte Carlo sampling method. For a given m, the data was generated and the error was calculated over multiple iterations. The average error over these iterations would therefore be the error for the given m. As expected, this method generated noisy results with few iterations, and more accurate results for more iterations.

As an alternative, and a way to corroborate the results of Monte Carlo sampling, we decided to test a method where the average error was computed iteratively. The errors were averaged over every value of m. The error that we are computing is the error averaged over all tested m values. If we are at $m = 10$, the error computed will be the cumulative error from $m = 1$ to $m = 10$.

Both methods were tested on the Winnow algorithm. Their performance is illustrated in Figure 12). Table 3 shows the computation time of every tested method.

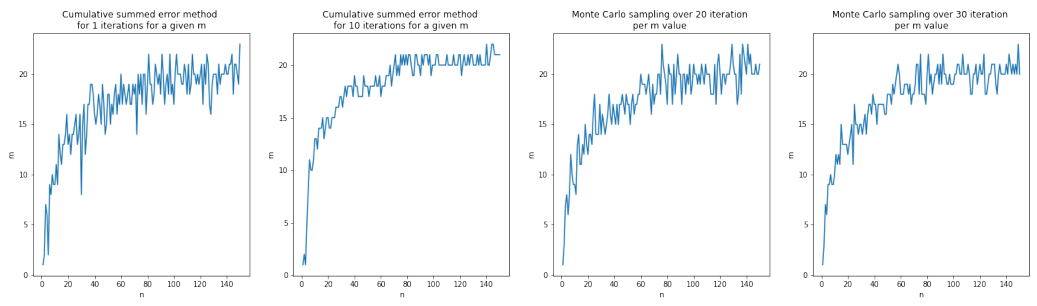

Figure 12: Results using different error calculation methods on the Winnow algorithm

Method used on the Winnow algorithm	**Computation Time (s)**
Monte Carlo sampling over 20 iteration per m value	44.51
Monte Carlo sampling over 30 iteration per m value	67.88
Cumulative summed error method for 1 iterations for a given m	22.75
Cumulative summed error method for 10 iterations for a given m	243.72

It can be observed that despite all curves being relatively noisy, the cumulative summed error method performs best. However, it requires the longest computation time. Taking into account the computation to noisiness trade-off, we were interested in the single iterative of the cumulative summed error method.

In order to cross validate our intuition we tested the method along with the Monte Carlo method for a high iteration count, on both perception and least squares algorithms. Given its considerably longer run time and stable results, the $1 - NN$ classifier was not considered in this evaluation. The final results are illustrated in figures 13 and 14, with computation times reported in table 3.

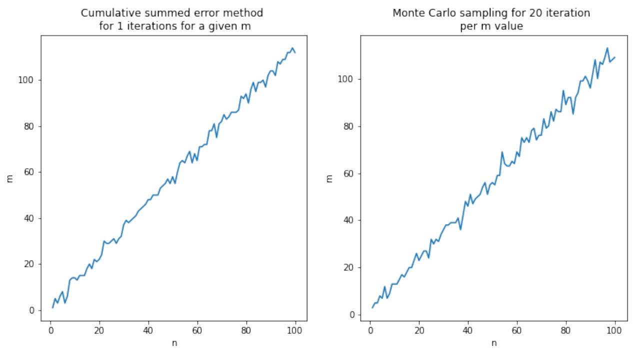

Figure 13: Comparative results using standard and cumulative error calculations on the perceptron algorithm

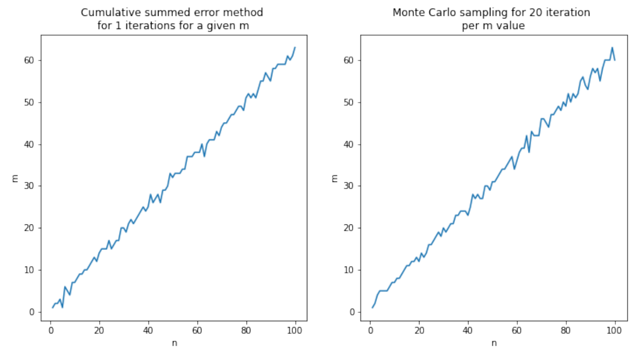

Figure 14: Comparative results using standard and cumulative error calculations on the least squares algorithm

Algorithm	**Method**	**Computation time (s)**
Perceptron	Cumulative sum, 1 iteration	270.02
Perceptron	Monte Carlo, 30 iterations	147
Least Squares	Cumulative sum, 1 iteration	113.13
Least Squares	Monte Carlo, 30 iterations	91.91

When testing the methods on the perceptron and least-squares algorithms, similar plots are returned. The Monte Carlo method resulted slightly noisier and faster. The cumulative summed error method returns a cleaner plot but requires more time to compute. The Monte Carlo method sample over multiple iterations. This will reduce bias. The cumulative error uses errors from past calculations. This would add bias to the model as the error calculations would have some dependence.

All in all, these 2 methods produce reasonable results, illustrating the sample complexities of these algorithms. The shapes of the observed curves are clearly distinguishable. We chose to obtain final estimates of sample complexity using the cumulative summed error method only because the returned plots were cleaner, and worth the extra computation time.

c.

All algorithms demonstrated different sample complexities, and therefore different estimated complexity functions:

- When plotting the sample complexity of the perceptron algorithm, the optimal training set size increases somewhat linearly as its dimension increases. Figure 8 demonstrates this relationship. We can the estimate that $O(n) = n$ for the perceptron algorithm.
- When plotting the sample complexity of the winnow algorithm, the figure illustrates a monotonically increasing, convex curve (9). While the curve slope is slowly decreasing, it does not seem to converge to a fixed value. We therefore assume that the curve is of logarithmic order. We thus estimate that $O(n) = log(n)$ for the winnow algorithm.
- When plotting the sample complexity of the least squares classification algorithm, the optimal training set size increases somewhat linearly as its dimension increases. Figure 10 demonstrates this relationship. We can therefore estimate that $O(n) = n$ for the least squares classification algorithm.
- When plotting the sample complexity of the $1 - NN$ algorithm, the optimal training set size increases somewhat exponentially as its dimension increases. Figure 11 demonstrates this relationship. We can the estimate that $O(n) = e^n$ for the $1 - NN$ algorithm.

d.

Proof Introduction Suppose we sample an integer $s \in \{1, ..., m\}$ uniformly at random. Derive a non-trivial upper bound $p^{m,n}$ on the probability that the perceptron will make a mistake on the s_{th} example after being trained on examples $(x_1, y_1), ..., (x_{s-1}, y_{s-1})$.

Lemmas

1 - Novikoffs Theorem For all sequences of examples,

$$\mathcal{S} = (x_1, y_1), ..., (x_m, y_m) \in \mathbb{R}^n \times \{-1, 1\}$$

the number of mistakes of the perceptron algorithm is bounded by,

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

where $R := max_t(x_t)$ and γ is the separation margin of the perceptron.

2 - Online to Batch Bounds Suppose the data is drawn from a distribution P and a mistake bound for algorithm A for any such set is B. Given m, then let t be drawn uniformly at random from $\{1, ..., m\}$. Let S consist of t examples sampled iid from P, let $(x\prime, y\prime)$ be an additional example sampled from P then,

$$Prob(\mathcal{A}_{\mathcal{S}}(x\prime) \neq y\prime) \leq \frac{B}{m},$$

with respect to the draw of t, S, and $(x\prime, y\prime)$.

Proof of Theorem In order to solve derive this non-trivial upper bound $p^{m,n}$ we will make use of both lemmas described above. We are sampling a number of points s uniformly at random from $\{1, ..., m\}$. Let $\mathcal{S}$ be the set of points of length s and $\mathcal{A}$ be the perceptron algorithm. We train $\mathcal{A}$ on points $(x_1, y_1), ..., (x_{s-1}, y_{s-1})$ and desire an upper bound on the probability that the perceptron will make a mistake on an additional example s. **Lemma 2** gives us a general upper bound for any algorithm. **Lemma 1** provides us with the upper bound for the perceptron algorithm. By combining both, we can derive $p^{m,n}$:

$$p^{m,n} \leq \frac{(R/\gamma)^2}{m}$$

Given that the maximum norm of the dataset is given by $R^2 = n$ we can say that $R = \sqrt{n}$. Additionally, given that all the data points are either 1 or -1, they will always be at a distance of $\gamma = 1$ from the optimal separating hyperplane. Hence, we derive:

$$p^{m,n} \leq \frac{n}{m}$$

e.

References

[1] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4), 2007.