COMP0078 Supervised Learning
Term 1 2021/22
Coursework 2

Team: Yes
Student Number: 21055399 , 20084999

December 13th, 2021

Contents

1. Part I

1.1 Kernel perceptron (Handwritten Digit Classification)

In the following sections, we are going to present a multi-class classifier that can classify hand written digits (in particular the MNIST dataset). The classifier will be trained with the kernel perceptron algorithm. The kernel would allow us to explore higher dimensional space with our data.

Kernel Perceptron for multiclass implementation

For binary kernel perceptron, the prediction formula is

$$\hat{y_t} = sign(\sum_{i=0}^{t-1} \alpha_i K(x_i, x_t)) \tag{1.1}$$

To expand the binary kernel perceptron to **k** classes trained with **m** datapoints. We want to train 10 classifier for each classes. Hence, we can initialise our weights matrix $\boldsymbol{\alpha}$ to be a k x m matrix.
The Prediction vector $\hat{y_t} \in \mathbb{R}^k$ becomes

$$\hat{y_t} = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,m} \end{bmatrix} \begin{bmatrix} K(x_1, x_t) \\ \vdots \\ K(x_m, x_t) \end{bmatrix}, \boldsymbol{\alpha} = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,m} \end{bmatrix} \tag{1.2}$$

Therefore, for every prediction $\hat{y_t}$, we will have a k $\times$ 1 vector and we can convert entries > 0 to +1, and -1 o/w.
Since our true label y_t is a real number, we will need to augment it so we do a valid comparison during weight update. We can convert it using one hot encoding to a k x 1 vector, meaning that if the true label is '5', +1 is only assigned to the fifth entry and all others will be -1. Then we can compare y_t with $\hat{y_t}$ entry by entry.
Finally, the update rules follows that we $+y_t$ for entries in the $\boldsymbol{\alpha}$ matrix where y_t and $\hat{y_t}$ differs.
There is similar operation for testing, since we are not doing 'online' learning we can expand the kernel to a matrix of size m $\times$ m_{test} where m_{test} is the number of testing samples. With this, the test prediction becomes

$$\hat{Y} = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,m} \\ \vdots & \ddots & \vdots \\ \alpha_{k,1} & \dots & \alpha_{k,m} \end{bmatrix} \begin{bmatrix} K(x_1, x_t) & \dots & K(x_1, x_{m_{test}}) \\ \vdots & \ddots & \vdots \\ K(x_m, x_t) & \dots & K(x_m, x_{m_{test}}) \end{bmatrix} \tag{1.3}$$

$\hat{Y}$ is a k$\times$m$_{test}$ matrix containing all the predictions for all testing samples with all k classifiers. We will need to convert $\hat{Y}$ to a 1$\times$m$_{test}$ vector by taking the argmax of each column. Then we can compare this with the true label to get the accuracy and error rates.

Question 1 - 5 classifier - One-vs-Rest

The first multi-class classifier that is implemented and used from Question 1 through 5 is the classic One-vs-Rest classifier. It involves splitting the multi-class dataset into multiple binary classification problems. We will then train a binary classifier on each binary classification problem and predictions are made using the model with the highest confidence. Since we have 10 digits (0 - 9) in the MNIST dataset, the One-vs-Rest classifier will have 10 models, each digit vs the rest. Since the dataset is split for one class vs all the rest, the number of data per dataset needs to include every data point. In Question 1 - 4, we will work with a simple polynomial kernel and in Question 5 we will move on to a Gaussian kernel.

Algorithm 1 One-vs-Rest multiclass classifier

Require: $Input : \{(x_1, y_1), \ldots, (x_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$

$\boldsymbol{\alpha} \leftarrow [0]_{k \times m}$
for t = 1 to m **do**
    $Receive : x_t \in \mathbb{R}^n$
    $Predict : \hat{y_t} = argmax_k(\sum_{i=1}^{m} \boldsymbol{\alpha}_i^{(k)} y_i K(x_i, x_t)$
    $Receive : y_t$
    if $\hat{y_t} y_t <= 0$ **then**
        $\boldsymbol{\alpha}_t^{(y_t)} = \boldsymbol{\alpha}_t^{(y_t)} + 1$
    else
        $\boldsymbol{\alpha}_t^{(y_t)} = \boldsymbol{\alpha}_t^{(y_t)} - 1$
    end if
end for

Question 6 classifier - One-vs-One

For Question 6, we have implemented the One-vs-One classifier. One-vs-One differs from one-vs-rest in that the one-vs-one approach splits the datasets into $\frac{10(10-1)}{2}$ datasets for each class versus every other class, instead of splitting it into one binary dataset for each class. So since we have 10 classes, the number of models we have would be $\frac{10(10-1)}{2} = 45$. This has significant more models compared to One-vs-rest (10). However, since the datasets are split for one class vs another class, the number of data per dataset (post split) is much smaller.

Algorithm 2 One-vs-One multiclass classifier

Require: $Input : \{(x_1, y_1), \ldots, (x_m, y_m)\} \in (\mathbb{R}^n, \{-1, 1\})^m$

$\boldsymbol{\alpha} \leftarrow [0]_{45 \times m}$
for t = 1 to m **do**
    Train each binary classifier following normal binary perceptron updates rule
end for
for t = 1 to m **do**
    $Receive : x_t \in \mathbb{R}^n$
    $Predict : \hat{y_t}^{(k)} = sign(\sum_{i=1}^{m} \boldsymbol{\alpha}_i^{(k)} y_i K(x_i, x_t))$ for k = 1 ... 45, i.e. each binary classifier.
    $Predict : \hat{y_t}$ = the class with the most votes
    $Receive : y_t$
    Incur train and test loss
end for

How we select the number of epochs

To determine the number of epochs in the experiments, we performed a number of initial tests on the kernel perceptron algorithm with polynomial kernel. We discovered that they generally converged after 6-7 epochs. To stay on the safe side we have selected a max epoch of 10. The reason why we set a max epoch is to prevent training indefinitely for non-linearly separable data when d=1,2 in the polynomial kernels.
On top of that, at each epoch, we will check for convergence using the previous train error and the current train error. If their difference is less than 0.001%, we will considered it as converged and break. We consider the tolerance to be tight enough that it shows good results.
Note, it is perfectly feasible to perform cross validation to tune the number of epochs and tolerance. However that would mean we have 3 hyperparameter and we would recommend tuning them in series instead of tuning through a multiple layers grid search.
* Consider the above online classifiers used in a batch setting.
** Consider all errors in the below sections as percentages instead of raw errors.

Question 1

In order to study the effect of the polynomial degree in the polynomial kernel, we present the mean and standard deviation for training error (train_err) and testing error(test_err) for each d value.

Testing protocol

We preform 20 runs for d = 1,...,7 and within each run the classifier is trained under 10 epochs or until convergence (i.e. tolerance between trn_err is less than 0.01).

Observations

(i) It is clear that all errors are very small except when d = 1. This show the non-linearity of the data set structure because the linear decision boundary clearly doesn't work well with our dataset.

(ii) There is a general down trend in the mean of the training and testing error when we increase the polynomial degree used in the polynomial kernel due to the more flexible decision boundaries of the higher polynomial degrees. Therefore the classifier will benefit from the non-linear boundaries and hence give a better fit to the dataset.

(iii) For d = 7, both the train and test error increase. It is not surprising for the test error to be higher as it is probably due to over-fitting. However, it is surprising to see the train error increases as well. One possibility is that we didn't train until convergence and we have slightly underfitted the data.

(iv) With a reasonable d (i.e. d=5) we achieve the lowest test error.

	d : 1	d : 2	d : 3	d : 4	d : 5	d : 6	d : 7
train_err_mean ± std	5.697±0.642	0.122±0.083	0.097±0.129	0.045±0.023	0.038±0.022	0.023±0.015	0.025±0.021
test_err_mean ± std	9.145±1.559	3.202±0.385	2.944±0.471	2.640±0.362	2.723±0.413	2.707±0.334	2.790±0.451

Figure 1.1: train error and test error mean and std

Question 2

To tune the polynomial degree d as a hyper-parameter, we use Cross-validation.

Results

mean d^* with std: 5.400±0.80
mean test error ± std: 2.755±0.462

Testing protocol

We preform 20 runs for d = 1,...,7 and use the 80% training data split to perform 5-fold cross-validation to find the optimal d^*in each run. Within each iteration of the 5-fold CV, the classifier is trained for less than 10 epochs or until convergence. Once we determine the d^* in every run, we retrain on the full 80% training set using d^* and record the test errors on the remaining 20% test dataset, which results to the following observations:

Observations

(i) The results are in line with the initial results we get from above. d values 5 and 6 seem to have the lowest mean test error. In addition, Mean test error for d=5 is 2.899, while test error for d=6 is 2.864.

(ii) From the mean d^* we know we will start overfitting when d > 5.

Question 3

To create a visualization of the performance of the classifier, we can create a confusion matrix. A confusion matrix is a specific table layout, where each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class.

Testing protocol

Per run:
To generate the confusion matrix, after we have obtained a d^* and retrained in the full training set in one of our 20 runs in Question 2, we will use the final weights to re test on the test data and subsequently obtain a confusion matrix. We calculate the rate by adding 1 to the confusion matrix mat$[y_t][\hat{y_t}]$ when we make a wrong prediction $\hat{y_t}$ and then divide it with the total number of y_t observed in the test set for each class.
Since we have 20 runs, we will have 20 confusion matrices at the end and thus we average out the results to get the average confusion error rate and its standard deviation.

Observations

(i) We start by mentioning that we can observe some very reasonable results. For example, a 5 could be confused as a 3, whereas an 8 could be confused for a 3 as well. These makes sense because 3, 5, and 8 have many similarities in their shape, for example the loop on the lower half of the digit. Therefore it is likely that the classifier has trouble distinguishing them.

(ii) The error rate is in its raw form, instead of a percentage.

Figure 1.2: Confusion matrix (heatmap)

Question 4

To further confirm where our classifier fails to perform, we have plotted the 5 hardest digits to predict as pixelated images.

Testing protocol

This part has similar protocol as Question 3.

Per run:
Once the classifer is retrained on the whole dataset using the d^* obtained in that run, we use the final weights in that run to test the performance on the entire dataset. We keep a counter for each test sample and the counter is incremented when the classifier prediction is different from the test label. Therefore, the hardest test samples to predict would be the ones that the classifier got wrong the most times.

Observations

(i) Again we can see some very reasonable results. All of the pixelated images are ill-written digits. The first three digits from the left look nothing like what their labels suggests.

(ii) As for the 8, it is ill-written as it is not as rounded on the top part as the other eights.

(iii) For the 5, it is missing a horizontal stroke, which makes it a very odd 5.

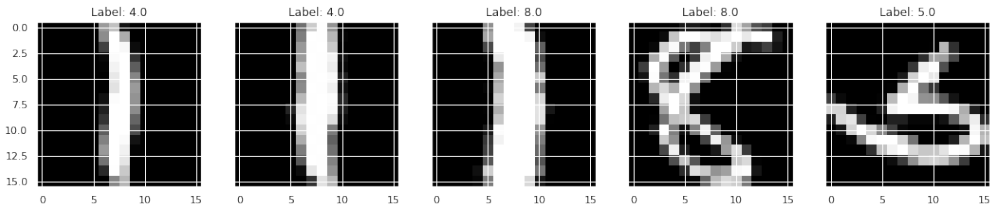


Figure 1.3: 5 hardest images to predict

Question 5.1

We now explore the effect of using a Gaussian kernel and repeat the steps done in Questions 1 and 2. At the end of Question 5.2, we will compare the Gaussian Kernel and the Polynomial kernel with the One-Vs-Rest Classifier.

Testing protocol

The testing protocol is more or less the same as Question 1. Except we have replaced the polynomial kernel with a gaussian kernel. And it has a hyperparameter c for us to tune. Since there is no suggested search range for the hyperparameter c in the gaussian kernel, some empirical experiments are performed. First, we try a broad range of c values specified as follows:

$$c\_val = [0.12500,\ 0.06250,\ 0.03125,\ 0.01562,\ 0.00781,\ 0.00391,\ 0.00195,\ 0.00098,\ 0.00049,\ 0.00024,\ 0.00012,\ 0.00006]$$

The results are shown in Figure 1.5. As we can see both the train loss and test loss are increasing after c=0.00781. Thus, an optimal c should be between 0.125 and 0.00781. Hence, an empirical binary search is performed multiple times and we end up with a final list for potential values of c:

$$c\_val = [0.1250,\ 0.0884,\ 0.0625,\ 0.0442,\ 0.0312,\ 0.0221,\ 0.0156,\ 0.0110,\ 0.0078,\ 0.0055]$$

Observations

(i) Compared to the polynomial kernel, the training error and testing error in the Gaussian Kernel is generally lower than that of polynomial kernel. This may be due to the fact that the gaussian kernel has an infinite number of dimensions in the feature space (since it can be expanded using an infinite Taylor Series). Therefore, the gaussian kernel is more flexible than the polynomial kernel and can give a better performance results compared to the polynomial kernel.

(ii) From Figure 1.6, we can see the lowest mean test error is obtained at c = 0.0156. The error is comparable if not slightly better than the result we get for the polynomial kernel. However, the result is not sufficiently better to allow us to conclude that we should use a radial basis function (RBF) kernel instead of a polynomial kernel.

	c: 0.12500	c: 0.06250	c: 0.03125	c: 0.01562	c: 0.00781	c: 0.00391	c: 0.00195	c: 0.00098	c: 0.00049	c: 0.00024	c: 0.00012	c: 0.00006
train_err_mean+/-std	3.326±4.231	2.873±3.625	1.752±2.793	1.136±2.302	1.389±2.646	2.355±3.137	4.526±3.621	7.464±3.798	10.328±4.499	13.356±5.748	16.730±7.883	20.866±10.691
test_err_mean+/-std	5.988±0.800	4.614±0.697	3.159±0.540	2.908±0.509	3.304±0.778	4.209±1.161	5.941±1.782	8.519±2.783	10.451±3.528	12.003±3.701	15.843±7.141	19.365±8.535

Figure 1.4: Initial experiment on rough value for c

	c: 0.1250	c: 0.0884	c: 0.0625	c: 0.0442	c: 0.0312	c: 0.0221	c: 0.0156	c: 0.0110	c: 0.0078	c: 0.0055
train_err_mean+/-std	0.006±0.012	0.015±0.017	0.015±0.015	0.019±0.016	0.018±0.012	0.030±0.022	0.026±0.017	0.040±0.022	0.053±0.027	0.124±0.074
test_err_mean+/-std	5.667±0.457	4.976±0.589	4.247±0.359	3.406±0.442	2.981±0.380	2.707±0.288	2.608±0.383	2.551±0.309	2.903±0.446	3.132±0.318

Figure 1.5: Train error and test error mean and std

Question 5.2

Results

mean test error ± std: 2.842±0.300
mean c^* with std: 0.017±0.004

Testing protocol

The testing protocol is again more or less the same as in Question 2. Except we have replaced the polynomial kernel with a gaussian kernel. We also use the same c values as in Question 5.1, that is we use:

$$c\_val = [0.1250,\ 0.0884,\ 0.0625,\ 0.0442,\ 0.0312,\ 0.0221,\ 0.0156,\ 0.0110,\ 0.0078,\ 0.0055]$$

Observations

(i) Similar observations as above hold here as well, the test error is lower using the gaussian kernel. This may once again be due to the fact that the gaussian kernel has an infinite dimensional feature space and allows the creation of arbitrarily flexible decision boundaries.

Question 6.1

Next we would like to see another method of generalizing a binary classification method to multi-class classification. Since we have already done One-vs-Rest, a natural choice would be One-vs-One. The difference has been outlined in the introduction of this part. A comparison between One-vs-One and One-vs-Rest will be presented in the observations part below.

Testing protocol/ implementation details

In order to prepare the dataset for One-vs-One classifier, first we will need to create a random dataset between each and every class, i.e. a dataset between class 0 and 1, class 0 and 2, class 0 and 3, etc.

Then we can perform 20 runs for each d ranging from 1 up to and including 7. In each run, we will train the $\frac{10(10-1)}{2} = 45$ distinct classifiers using the respective train data set for each class combination (for example class 0 and 1) for 10 epochs each or until convergence. For each binary classifier we will train for a maximum of 10 epochs or until convergence using the train loss for the binary classifier.

After training the 45 binary classifiers we will perform prediction on the full train and test dataset. If a binary classifier of classes 0 and 1 predicts a test point x^* to be in class 1, then one vote for class 1 for test point x^* would be recorded. Finally, the class that has the most votes for each test and train point would be the final test and train predictions respectively.

Observations

(i) Compared to One-vs-Rest, the overall mean train and test errors are higher except for d = 1. The reason why test and train errors are lower for d = 1 compared to One-vs-Rest is because there is a higher chance for each binary classifier to separate the binary datasets. Hence when we combine the results of the 45 classifiers the overall performance is better.

(ii) The overall mean train errors are higher because the 45 classifiers are trained using only part of the full training set. And since the class specific dataset is much smaller, there is a higher chance that the

binary classifiers to overfit on the subset of the full training set. Hence when we calculate the overall train error it doesn't generalise as well.

(iii) The overall mean test errors are higher because 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 results in a higher error in the test data.

(iv) Furthermore, compared to One-vs-Rest, the optimal hyperparameter d^* is smaller when using One-vs-One, but its test error is higher. One explanation is for One-vs-One we have more classifiers in total and hence the lack of generalisation in the boundaries for a lower polynomial order is compensated with the generalisation in the boundaries from the combined effect of multiple classifiers. However since the post processed dataset is smaller, each binary classifer also has a higher chance of overfitting to the training data, and hence a higher test error is observed.

	d : 1	d : 2	d : 3	d : 4	d : 5	d : 6	d : 7
train_err_mean ± std	3.042±0.842	0.385±0.169	0.195±0.059	0.156±0.074	0.159±0.083	0.096±0.049	0.122±0.155
test_err_mean ± std	6.417±0.828	3.624±0.340	3.387±0.422	3.430±0.292	3.511±0.305	3.312±0.490	3.527±0.480

Figure 1.6: Train error and test error mean and std

Question 6.2

Results

mean test error ± std: 3.511±0.306
mean d^* with std: 4.250±0.887

Observations

Identical observations to those previously mentioned in Question 6.1 also hold here.

Conclusion

After evaluation, we see the effect of different kernels results in different performance and optimal hyperparameter. However, the result does not support a statement of superiority between kernels and we should always consider multiple kernels when training classifier.
Two different multiclass classifier methods, One-vs-Rest and One-vs-One are evaluated using kernel perceptron. One-vs-Rest might result in a lower testing error. However it has its drawback in the imbalance in training data. The binary classification learners see unbalanced distributions because typically the set of negatives they see is much larger than the set of positives.
Therefore, this study leaves a lot of room for the study of other classification methods such as KNN and SVM and by no means a comprehensive comparison between different kernels and methods. They each have their merits and should be considered depending on the dataset.

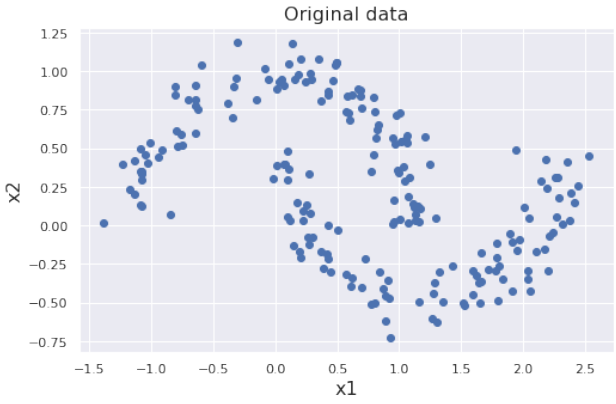
2. Part II

2.1 Spectral Clustering

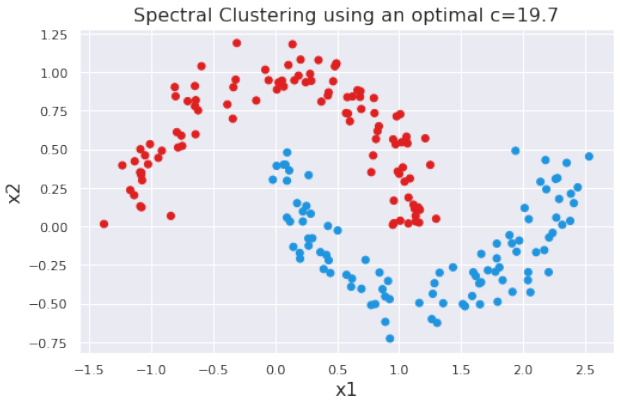
Experiments

We begin this section with short description of the spectral clustering implementation in the code. Firstly, we create the function 'gaussian weight' which is used to generate the weight matrix of the dataset. Then, the function 'calc laplacian' generates the Laplacian matrix by following the description of the assignment. Lastly, the final function to complete the spectral clustering implementation is 'spectral clustering' which essentially assigns cluster labels to the datapoints by analysing the entries of the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix.

1. Below are two plots, one showing the original data, and the other showing the results of the spectral clustering algorithm using one of the values of c which correctly clusters all of the data. With regards to choosing an optimal value of c, we note that as shown in the code, there are a number of distinct values which give perfect clustering results but we simply choose the smallest one which is roughly equal to 19.7.



(a) Plot of the twomoons data

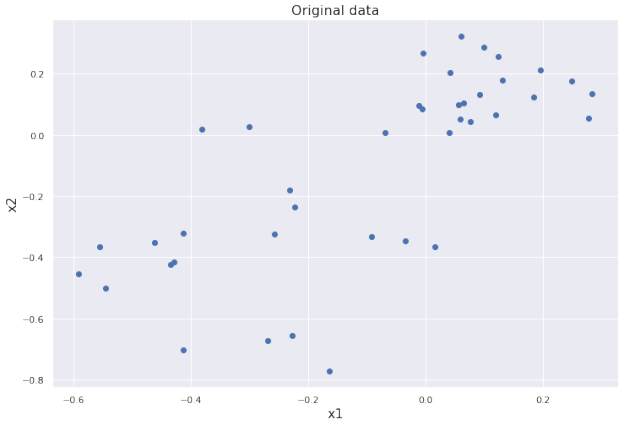


(b) Spectral clustering on the twomoons dataset using an optimal value of c

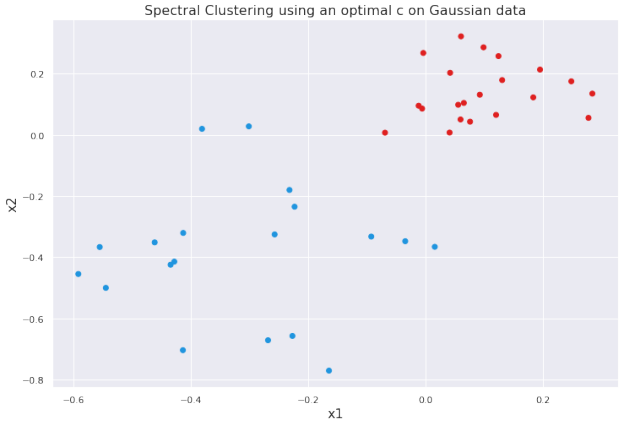
Figure 2.1: Twomoons dataset experiments

2. For this part we apply spectral clustering to data generated from two distinct Gaussian distributions as described in the assignment. For this data, the spectral clustering algorithm gives nearly perfect results, with a correct cluster percentage of 97.5% using the smallest of all such c values which is equal to 34.3.

3. Finally, we apply the spectral clustering algorithm on the dtrain123 data using a range of [0, 0.10] for the values of c. The value of c that gives the best results in this range is 0.012, and for all the greater c values the correct cluster percentage remains almost unchanged.



(a) Plot of the data coming from two distinct Gaussian distributions



(b) Spectral clustering on the Gaussian dataset using an optimal value of c

Figure 2.2: Gaussian data experiments

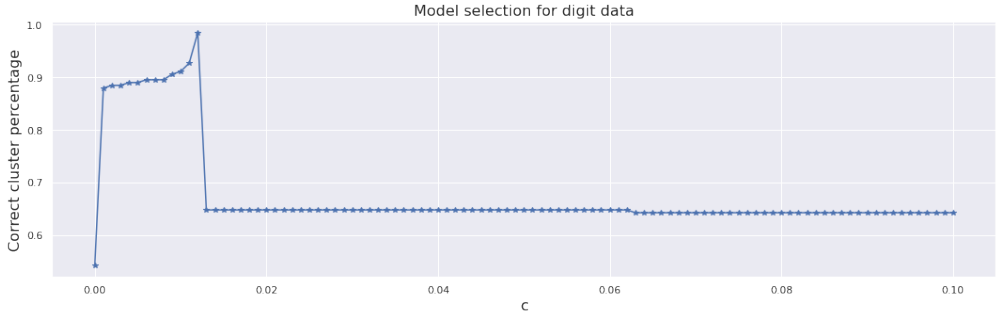


Figure 2.3: Correct cluster percentage versus c value on the digit data

Questions

1. In essence, a good clustering algorithm should be successful at being able to distinguish between the different subgroups within the data (equivalently the clusters). In contrast to a classification algorithm, we are not really concerned with predicting the 'correct' cluster label, we are rather interested in assigning different cluster labels to the distinct subgroups. Thus, whether our algorithm assigns a cluster label of +1 or -1 to a particular group of points is of little importance, as long as those cluster labels are assigned consistently. In the two digit case, the correct cluster percentage can be interpreted as the maximum percentage of data points assigned to the correct cluster group, and renders it a suitable metric for evaluating the quality of a clustering algorithm.

2. We begin by stating a well-known result, graph Laplacians are positive semi-definite matrices and thus all of their eigenvalues are non-negative. The fact that 0 is an eigenvalue of any Laplacian follows directly from the way the Laplacian matrix is constructed, which is that its columns are linearly dependent. For example, adding all of the columns (shown below) gives the zero vector. Hence, the vector of ones is an eigenvector of eigenvalue 0. Using the aforementioned result about positive semi-definite matrices it has to be that 0 is the smallest eigenvalue of the graph Laplacian. Below we prove that any constant vector is an eigenvector of eigenvalue 0 of the graph Laplacian. Consider, the constant vector $\mathbf{u} := \lambda\mathbf{1} \in \mathbb{R}^l$, where $\mathbf{1}$ is a vector of ones, and for some $\lambda \in \mathbb{R}\backslash\{0\}$. We show that $\mathbf{u}$ is an eigenvector of eigenvalue zero of the Laplacian:

$$L\mathbf{u} = (D - W)\mathbf{u} = \lambda(D - W)\mathbf{1} = \lambda(D\mathbf{1} - W\mathbf{1})$$

$$= \lambda \times \begin{bmatrix} \sum_{i=1}^{l} W_{1i} & 0 & \dots & 0 \\ 0 & \sum_{i=1}^{l} W_{2i} & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & \sum_{i=1}^{l} W_{li} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} - \begin{bmatrix} W_{11} & \dots & W_{1l} \\ \vdots & \ddots & \vdots \\ W_{l1} & \dots & W_{ll} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

$$= \lambda \times \begin{bmatrix} \sum_{i=1}^{l} W_{1i} \\ \vdots \\ \sum_{i=1}^{l} W_{li} \end{bmatrix} - \begin{bmatrix} \sum_{i=1}^{l} W_{1i} \\ \vdots \\ \sum_{i=1}^{l} W_{li} \end{bmatrix} = \lambda\mathbf{0} = \mathbf{0}$$

   Hence, the constant vector $\mathbf{u}$ is an eigenvector of the Laplacian of eigenvalue zero, which proves that the first eigenvalue of the Laplacian is zero.

3. Given a graph representation of the data, the aim of spectral clustering is to effectively partition the graph into two groups in our case, which correspond to the two clusters. The motivation behind spectral clustering is that we aim to partition the graph in a way such that the weights on the edges of vertices connecting the two distinct groups are small, and the weights on the edges of vertices within a group are large. This proves to be a very natural way to do so as we are interested in partitioning the data according to some measure of similarity between the datapoints on the graph. To achieve this, spectral clustering aims to efficiently solve the relaxed balanced cuts problem seen in the lecture slides,

shown in figure 2.4 below. It does so by providing an approximate (in the sense that each entry of the solution vector is allowed to be any real number) minimizer to this problem by finding the eigenvector corresponding to the second smallest eigenvalue of the graph Laplacian matrix. Finally, in our case, using the sign function on each entry of this eigenvector provides a way to partition the entries of the solution vector in two distinct groups which form the two clusters of the data.

$$\min_{\mathbf{u}} \mathbf{u}^\top \mathbf{L}\mathbf{u} \text{ s.t. } \quad \mathbf{u} \in \mathbb{R}^n, \quad \mathbf{u} \perp \mathbf{1}, \quad \|\mathbf{u}\|^2 = n$$

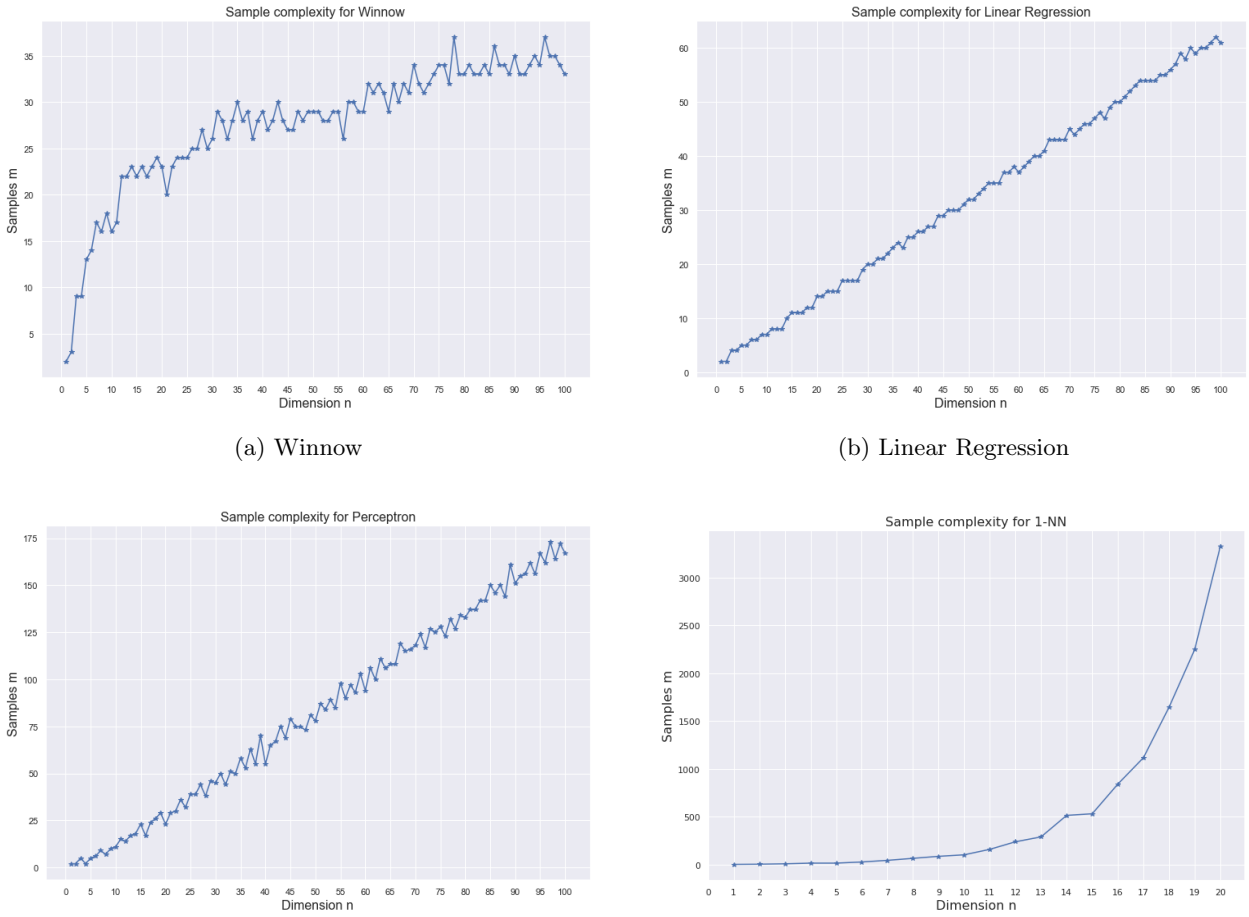
Figure 2.4: Balanced cuts objective for spectral clustering

4. As $c \to \infty$ the entries of the weight matrix tend to zero and hence the Laplacian matrix tends to the zero matrix. In this case, in a trivial sense the only eigenvalue is zero (repeated l times) and the set of eigenvectors is the standard Euclidean basis of $\mathbb{R}^l$. Hence, no matter which eigenvector we pick from the eigenbasis for v_2, the entries will be all either 0 or 1, and hence we always assign a +1. Moving to the other extreme, as $c \to 0$ the entries of the weight matrix tend to 1 and the Laplacian matrix tends to a matrix with $l-1$ on the diagonal and -1 elsewhere. The eigenvalues of this matrix are l (repeated $l-1$ times) and 0. Hence, in this case we will set v_2 to be an eigenvector corresponding to the second smallest eigenvalue, which here is zero. Its corresponding eigenvector is any constant vector (as shown above) and thus the cluster labels will be all +1 or all -1. In both extreme cases, we assign all data points to a single cluster. As a result, the quality of the clustering is very bad for very large and very small values of c. For intermediate values of c, when c is relatively small, the weight matrix entries become bigger and hence encourage the predicted cluster labels to be similar. The opposite follows for c relatively large via a similar argument. However, as we can see in the experiments section, when considering intermediate values of c, it is not obvious what the effect on the quality of clustering is. In general, c influences the quality of clustering as it changes how 'close' different points are on the graph, through the weights between them.

3. Part III

3.1 Questions

(a) In this part we implement the four different algorithms discussed in the question and provide their respective sample complexity plots in Figure 3.1 below. Observations and discussion about their performance follows in subsequent question parts.



(a) Winnow

(b) Linear Regression

(c) Perceptron

(d) 1-NN

Figure 3.1:
Sample complexity plots for the four algorithms

(b) Since computing the sample complexity exactly is of exponential complexity, it is impossible to compute it for large values of n and we need to resort to methods for accurately estimating it. To firstly estimate generalization error, we use the average test error (i.e. proportion of misclassified test points) over a number of different test sets (each model trained on a different training set) of size 300 datapoints

each. More specifically, for each dimension of the data considered, we increase the number of training data points used to fit each model until we reach a point where the average test error as described above is no more than 10%. When this condition is satisfied, we take the smallest number of training data points required to get an average generalization error of no more than 10%, and use this number as an estimate of the sample complexity for this particular dimension. The rationale for taking this approach is motivated by the law of large numbers, which states that the average taken over a large number of trials should closely approximate the expected value. Thus, the law of large numbers is very relevant to our use of the average test error as an estimator for the generalization error, which is necessary to estimate sample complexity.

Moving on to the second part of this question, the main trade-off in this approach is between the accuracy of the sample complexity estimate and the computational time. More specifically, to improve the accuracy of our sample complexity estimate, we need to estimate the generalization error with more accuracy as well. This requires reducing the standard deviation of our sample mean estimate, which decreases at a rate of $\sqrt{n}$, where n is the number of test sets we average over. For computation time purposes, we restrict our generalization error estimates to averaging over 50 different test sets for all models except 1-NN, for which we only use 5 due to its exponential sample complexity resulting from the curse of dimensionality.

With regards to the biases of this method, using only 15 different test sets might yield inaccurate estimates for sample complexity. We also note that a drawback of this method is that we do not know the uncertainty in our estimates, which we could obtain at a significant computational cost by repeating the process many times and calculating standard deviation estimates. Last but not least, as the dimension n increases, our generalization error estimate becomes worse since the number of datapoints required to exactly compute the generalization error becomes enormous (we need to sum over 2^n terms). As a result, we expect that the quality of our sample complexity estimates decreases as the dimension n increases.

(c) For this part of the question, we base our discussion regarding the behaviour of the sample complexity as dimensionality of the data tends to infinity based on the experimental observations in figure 3.2. We begin by noting that sample complexity of the linear regression and perceptron algorithms grows linearly with dimension, and is easily seen to be in $\Theta(n)$. Taking the linear regression case to illustrate this, the result immediately follows by setting $c_1 = 0.5, c_2 = 0.75$ and $n_0 = 20$ in the standard Big Theta notation definition. In other words, for all $n > n_0$, the sample complexity plot for linear regression is expected to lie between the $m = 0.5n$ and $m = 0.75n$ lines. Moving to the Winnow algorithm, following the same logic, we can see that sample complexity grows logarithmically as a function of dimension, and hence is in $\Theta(log(n))$. To make things more concrete, we refer to plot (e) in figure 3.2 and see that the linear regression model of m versus log(n) is extremely statistically significant, as indicated by the F-statistic p-value (essentially zero) and the very high R-squared value. This analysis confirms our experimental observations regarding the logarithmic sample complexity of the winnow algorithm. A similar argument can be made for the 1-NN algorithm based on the plot below, and a regression analysis argument (found in the Jupyter notebook) as in the winnow case. From these, it follows that sample complexity grows exponentially as a function of dimension, which gives that sample complexity of 1-NN is in $\Theta(c^n)$, for some $c > 1$. This is something we should be expecting, given that methods like KNN suffer from the curse of dimensionality.

Finally, we compare the performance of the four algorithms in figure 3.3 and observe that as expected from the above analysis, the order of performance (from best to worst) of the four algorithms when it comes to increasingly large dimensions is: Winnow, Linear Regression, Perceptron and 1-NN. However, an interesting feature of the plot is that linear regression performs better in terms of sample complexity compared to winnow for data dimensions of until around 45.

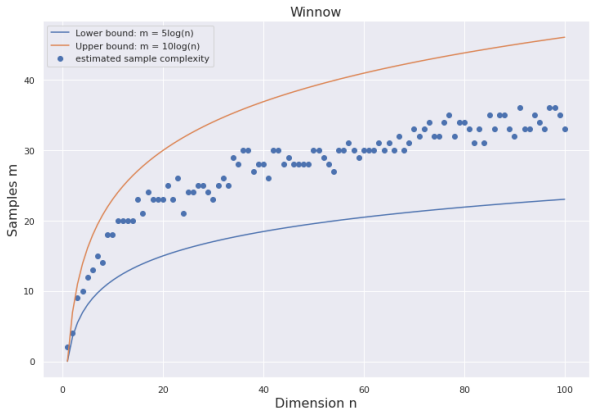
(d) To derive a non-trivial upper bound for the probability in the question we consider the general case where the data points lie in $\mathbb{R}^n$, for some $n \in \mathbf{N}$. We firstly note that since any entry in the data points is either 1 or -1, we have $\|x_i\| = \sqrt{n}, \forall x_i \in \mathcal{S}_m$. Thus, $\|x_i\| \leq \sqrt{n}, \forall x_i \in \mathcal{S}_m$. Moving on, we observe that the data in this particular dataset are linearly separable by noting that points belonging to the same class, share the same first coordinate. With this in mind, it's easy to see that the $n-1$ dimensional hyperplane defined by $x_1 = 0$ perfectly separates the two classes, since each class lies on one of the two sides of the plane. We further proceed and argue that the data are linearly separable by a margin of one. To see this, consider some point $x = (x_1, x_2, \ldots, x_n)$, then the point $x' = (0, x_2, \ldots, x_n)$ lies on the separating hyperplane, and the distance (equivalently the margin) between points x and x' is easily seen to be equal to $\sqrt{x_1^2} = \sqrt{1} = 1$. Using the Novikoff Perceptron mistake bound theorem from the notes with $R = \sqrt{n}$ and $\gamma = 1$, we conclude that with our data, the perceptron can make at most $(\frac{\sqrt{n}}{1})^2 = n$ mistakes.

With the above bound in hand, we proceed and directly apply the Theorem for using online bounds to give bounds in batch settings from the Online Learning slides (slide 59 in particular), which gives:

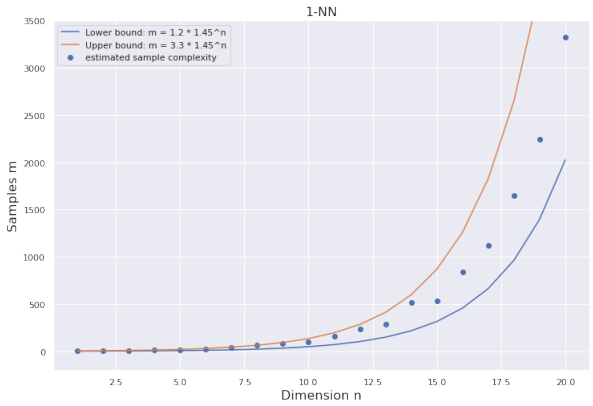
$$\hat{p}_{m,n} \leq \min\left(\frac{n}{m}, 1\right)$$

by also mentioning an obvious fact, probabilities can be at most one.

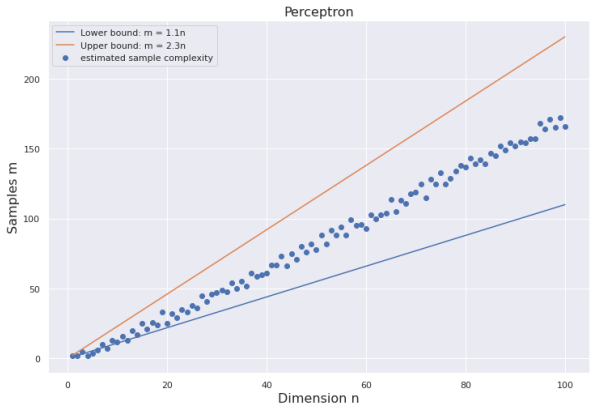
(e)



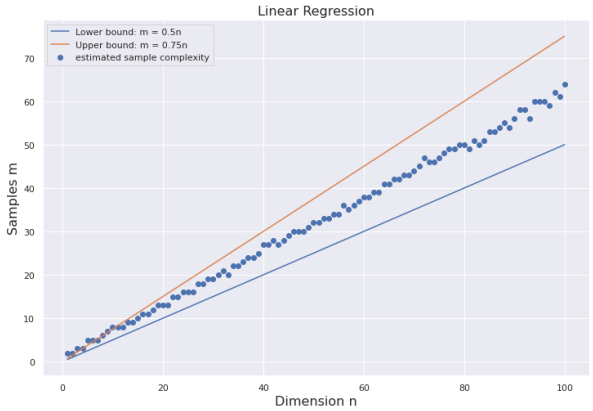
(a) Winnow



(b) 1-NN



(c) Perceptron



(d) Linear Regression

```
mod = sm.OLS(m_winnow, np.log(n)).fit()
mod.summary()
```

OLS Regression Results

Dep. Variable:	y	**R-squared (uncentered):**	0.997
Model:	OLS	**Adj. R-squared (uncentered):**	0.997
Method:	Least Squares	**F-statistic:**	3.436e+04
Date:	Wed, 08 Dec 2021	**Prob (F-statistic):**	1.23e-127
Time:	13:59:05	**Log-Likelihood:**	-183.82
No. Observations:	100	**AIC:**	369.6
Df Residuals:	99	**BIC:**	372.2
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
x1	7.5498	0.041	185.366	0.000	7.469	7.631

Omnibus:	3.433	**Durbin-Watson:**	1.290
Prob(Omnibus):	0.180	**Jarque-Bera (JB):**	2.968
Skew:	-0.257	**Prob(JB):**	0.227
Kurtosis:	3.669	**Cond. No.**	1.00

(e) Regression analysis for Winnow

Figure 3.2: Experimental average sample complexity plots and regression analysis for the four algorithms

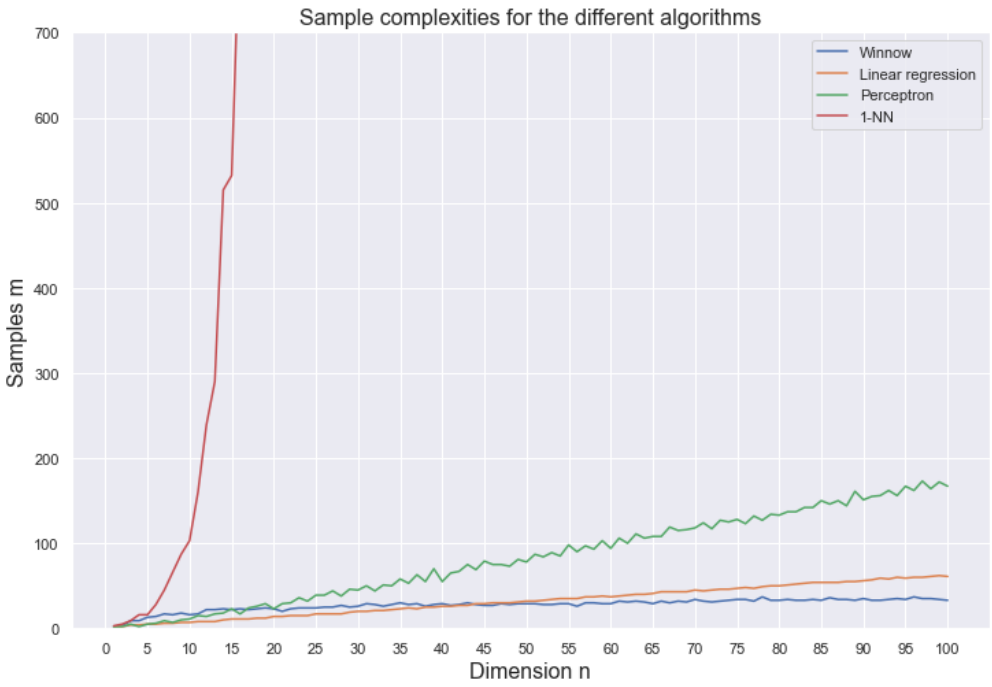


Figure 3.3: Sample complexity versus dimension plot