Supervised Learning (COMP0078) - Coursework 2

Group: 17112496 and 17014657

December 20, 2021

1 Part I

1.1 Kernel Perceptron (Handwritten Digit classification)

1.1.1 Introduction

We are going to experiment different ways to do multi-class classification with handwritten digit recognition dataset that has 10 different labels: from 0 to 9. We will use 'Perceptron' as our main algorithm that will classify 10 handwritten digits. However, since perceptron is a binary classification model, it is impossible to use naive perceptron methodology in a multi-class classification situation.
We will look into two ways that make it possible: 'One Vs. Rest' and 'One Vs. One'.

Not only that, we will apply two different kernels (Polynomial Kernel and Gaussian Kernel) into our algorithm to map the data space into a higher and a more linearly separable dimension.

Polynomial Kernel is defined as:
$K_d(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i} \cdot \boldsymbol{x_j})^d$

Algorithm 1 Apply Polynomial Kernel

return $DOT(X_i, X_j)^d$

d is a kernel hyperparameter that we will cross-validate over for finding an optimal parameter.

Gaussian Kernel is defined as:
$K(\boldsymbol{x_i}, \boldsymbol{x_j}) = e^{-c\|\boldsymbol{x_i} - \boldsymbol{x_j}\|^2}$

Algorithm 2 Apply Gaussian Kernel

$I \leftarrow SUM(X_i^2)$
$J \leftarrow SUM(X_j^2)$
$CompleteSquared \leftarrow I + J - 2 * DOT(X_i, X_j)$
return $EXP(-c * CompleteSquared)$

c, the width of a kernel, is a kernel hyperparameter that we will cross-validate over for finding an optimal parameter.

Both Polynomial and Gaussian Kernels are valid Kernels

1.1.2 One Vs. Rest

As mentioned above, we need to generalize the bianary classification model to K-class classifier. One method of generalization is called One versus Rest.

If we have K classes, then we make K binary classifiers. Each classifier will only classify if the data point belongs to its class. If the data point belongs to its class, then it will classify as 1, and -1 otherwise. Therefore, to train these K classifiers, we need to create K training datasets with appropriate sign values as labels.

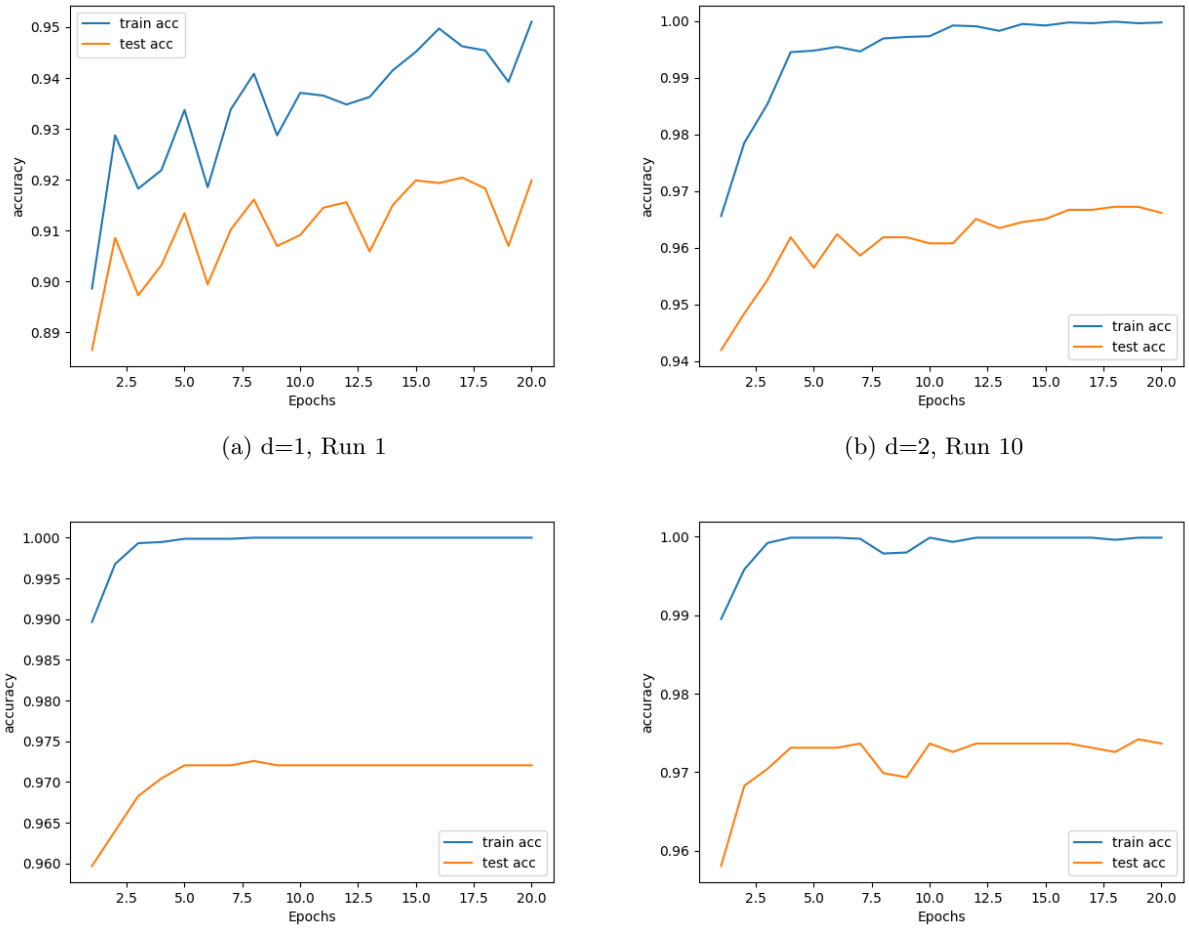
1.1.3 One Vs. One

In this method, if we have K classes, then we make $K(K-1)/2$ binary classifiers. This is because we make every combinations for each classes as ${}_kC_2 = \frac{k!}{2!(k-2)!}$. In other words, we split the primary dataset into one dataset for each class opposite to every other class. Each binary classifier predicts one class. Since it has to create much more classifiers compared to One Vs. Rest method, it is computationally costly.

1.1.4 Six Experiments

1. From experiment number 1 to 4, we will use polynomial Kernel with One Vs. Rest Perceptron. From this experiment, we perform 20 runs for $d = 1, ..., 7$ while at each run, dataset is randomly shuffle splitted into 80% training set and the other 20% testing set. Here, we will record mean test and train errors with their standard deviations.

   Before we begin, epoch size is set to 20, because it's nearly the step when the graph of train and test accuracy plateaus as Figure 1 shows. Also, Table 1 shows the mean and standard deviations of training and testing errors.



(a) d=1, Run 1

(b) d=2, Run 10

(c) d=5, Run 13

(d) d=7, Run 4

Figure 1: Train and Test data accuracy over epochs

$d =$	Training error (%)	Testing error (%)
1	0.100 ± 0.021	0.114 ± 0.021
2	0.036 ± 0.007	0.057 ± 0.009
3	0.021 ± 0.004	0.048 ± 0.006
4	0.016 ± 0.003	0.043 ± 0.005
5	0.015 ± 0.007	0.043 ± 0.008
6	0.017 ± 0.024	0.046 ± 0.026
7	0.010 ± 0.001	0.039 ± 0.004

2. From experiment 2 to 4, we will perform 20 runs and produce the final result in the same script file. Here, in experiment 2, we do cross validation to select the best kernel parameter value d*. Then, by using the best parameter d*, we will retrain the 'One Vs. Rest' model with the full 80% training set. Table 2 shows the testing error and d* per run.

Run	Testing error (%)	d*
1	0.029	6
2	0.033	5
3	0.032	5
4	0.044	5
5	0.034	5
6	0.038	5
7	0.031	7
8	0.031	5
9	0.034	5
10	0.039	6
11	0.035	5
12	0.040	5
13	0.040	6
14	0.030	6
15	0.045	5
16	0.032	6
17	0.037	5
18	0.033	6
19	0.040	5
20	0.034	6

Table 2 shows the overall mean and standard deviation of the result.

Mean and STD of testing error	0.036 ± 0.004
Mean and STD of d*	5.45 ± 0.589

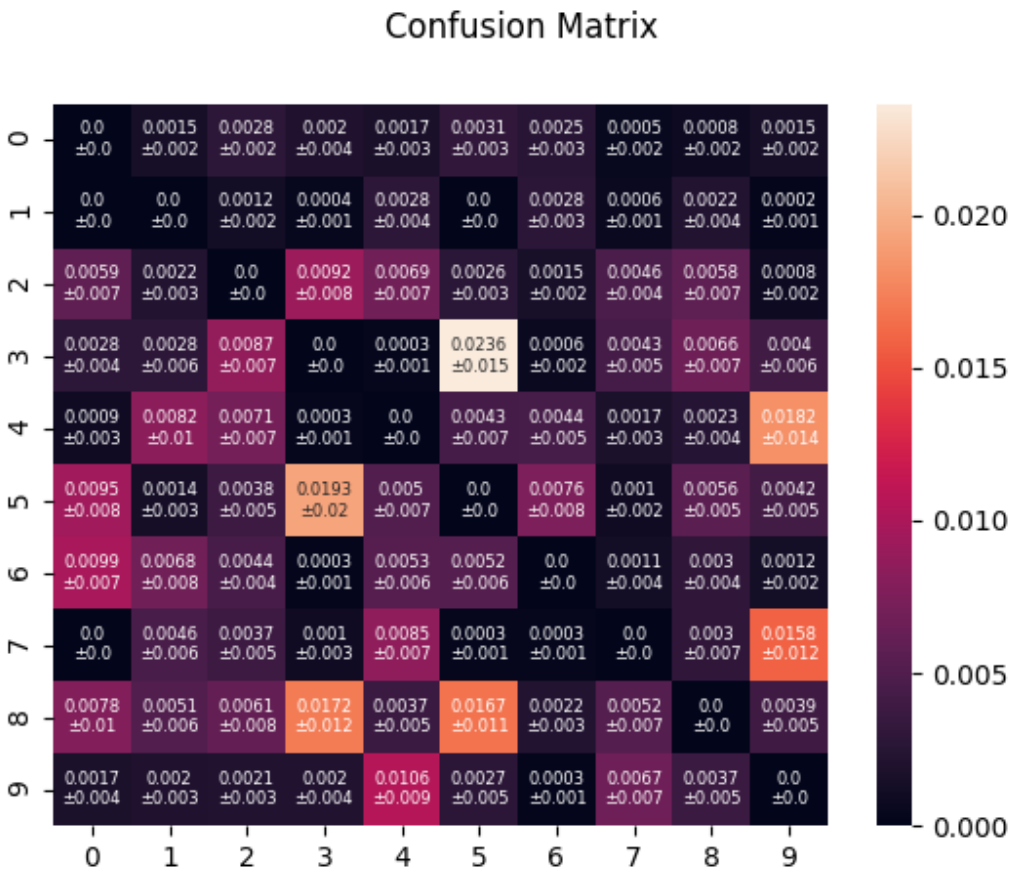
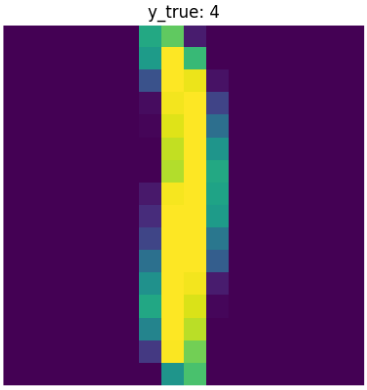
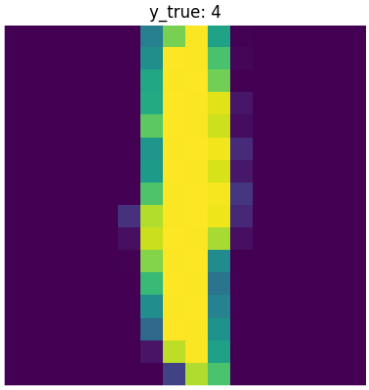


Figure 2: Confusion Matrix formed after retrained with d*

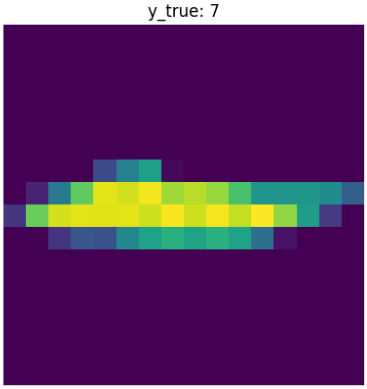
3.



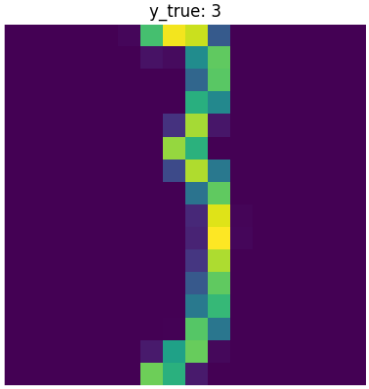
(a) First hardest



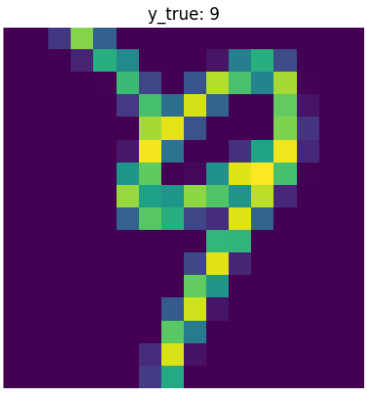
(b) Second hardest



(c) Third hardest



(d) Fourth hardest



(e) Fifth hardest

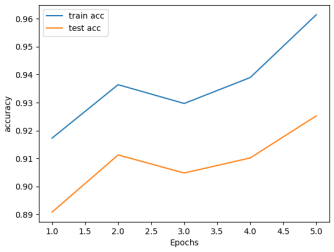
Figure 3: Five hardest digits to predict

4.

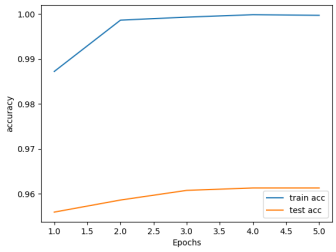
5. In this experiment, we will use Gaussian Kernel as our kernel function. Remember the parameter was c. Before we repeat the experiment 1 and 2, we need to do some initial search for the range of reasonable c. Like we have done in experiment 1, we will plot train and test accuracy graphs to diagnose if we are in a reasonable range of c value.

Trial 1

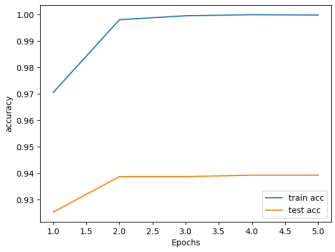
```
import numpy as np

# [0.001, 0.05623413251903491, 3.1622776601683795, 177.82794100389228, 10000.0]
C = np.logspace(-3, 4, 5)
```



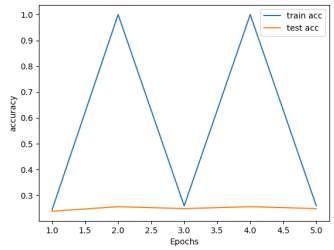
(a) trial 1 c=0.001



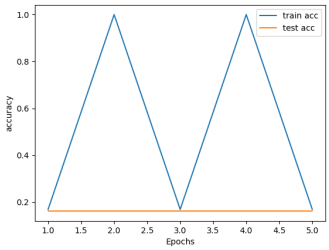
(b) trial 1 c=0.056



(c) trial 1 c=3.16



(d) trial 1 c=177



(e) trial 1 c=10000

Figure 4: Initial C search: Trial 1

From trial 1 we could see that the model starts to fluctuates heavily when c=177 and c=10000. We do not include these in next trial.

Trial 2

```
# [0.001, 0.005623413251903491, 0.03162277660168379, 0.1778279410038923, 1.0]
C = np.logspace(-3, 4, 5)
```

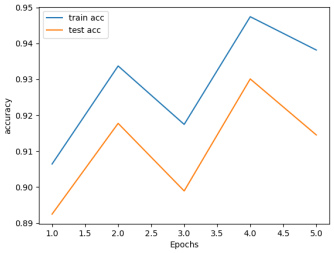
Now that we found an approximate upperbound of c value from Figure 5, we now want to see how the model behaves when c is less than 0.001

Trial 3

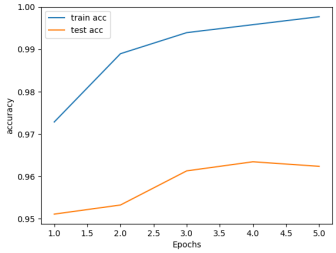
```
C = [0.0005, 0.001, 0.005, 0.007, 0.008, 0.009]
```

From Trial 3, we can learn that if the c value is less than 0.001, we see that test accuracy is higher than the train accuracy. The model is optimised on training data but if the accuracy comes higher is testing data, we know there's something wrong with this parameter.

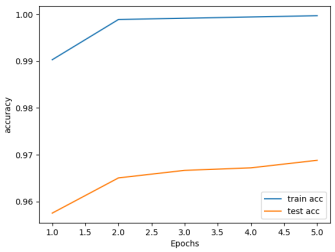
Now, The result of the repetition of experiment 1 from Trial 3 is below in Table 5:



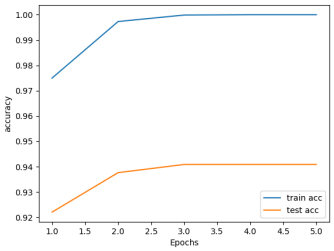
(a) trial 2 c=0.001



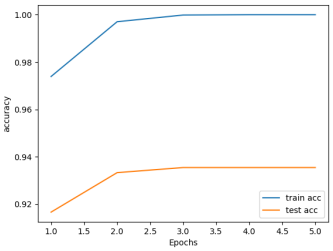
(b) trial 2 c=0.005



(c) trial 2 c=0.031

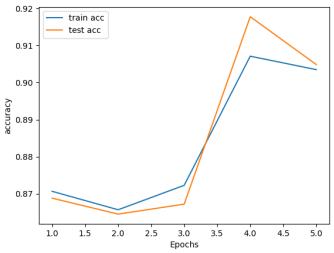


(d) trial 2 c=0.177

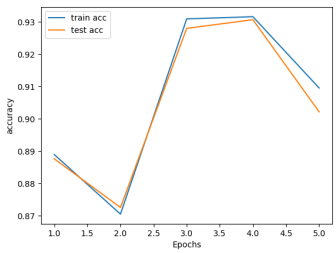


(e) trial 2 c=1.0

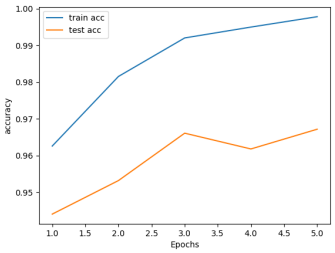
Figure 5: Initial C search: Trial 2



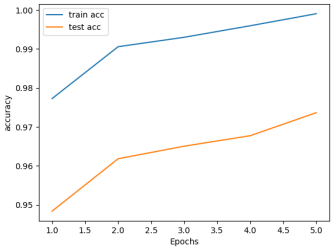
(a) trial 3 c=0.0005



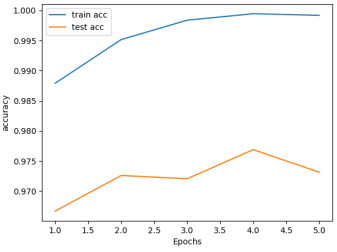
(b) trial 3 c=0.001



(c) trial 3 c=0.005



(d) trial 3 c=0.007



(e) trial 3 c=0.009

Figure 6: Initial C search: Trial 3

$c =$	Training error (%)	Testing error (%)
0.0005	0.09 ± 0.0071	0.100 ± 0.007
0.001	0.07 ± 0.02	0.09 ± 0.016
0.005	0.003 ± 0.002	0.037 ± 0.005
0.007	0.001 ± 0.001	0.029 ± 0.002
0.008	0.002 ± 0.001	0.032 ± 0.005
0.009	0.0005 ± 0.0003	0.032 ± 0.004

Now, the replication of the experiment 2 with Gaussian Kernel can be seen in Table 5 and 5.

Run	Testing error (%)	c*
1	0.029	0.007
2	0.034	0.009
3	0.023	0.009
4	0.036	0.01
5	0.032	0.008
6	0.029	0.01
7	0.027	0.009
8	0.034	0.01
9	0.034	0.01
10	0.025	0.009
11	0.027	0.008
12	0.030	0.01
13	0.027	0.01
14	0.023	0.009
15	0.033	0.01
16	0.0306	0.01
17	0.0413	0.009
18	0.027	0.01
19	0.025	0.01
20	0.031	0.01

Table 5 shows the overall mean and standard deviation of the result.

Mean and STD of testing error	0.030 ± 0.004
Mean and STD of c*	0.009 ± 0.0008

6. In experiment 6, we will replace 'One Vs. Rest' with 'One Vs. One'.
   Note the we use Polynomial Kernel again here.

   The result of the repetition of experiment 1 is below in Table 6:

$d =$	Training error (%)	Testing error (%)
1	0.014 ± 0.003	0.058 ± 0.003
2	0.0003 ± 0.0004	0.035 ± 0.002
3	0.0002 ± 0.0002	0.032 ± 0.005
4	0.0001 ± 0.0001	0.0308 ± 0.004
5	0.0002 ± 0.0002	0.0336 ± 0.003
6	0.0004 ± 0.001	0.035 ± 0.004
7	0.0001 ± 0.0001	0.033 ± 0.003

   Now, the replication of the experiment 2 with Gaussian Kernel can be seen in Table 6 and 6.

Run	Testing error (%)	d*
1	0.034	6
2	0.037	6
3	0.040	4
4	0.037	4
5	0.048	4
6	0.043	7
7	0.037	5
8	0.038	5
9	0.037	5
10	0.047	5
11	0.044	5
12	0.046	4
13	0.037	4
14	0.042	7
15	0.0413	7
16	0.044	5
17	0.0467	5
18	0.0413	5
19	0.0413	4
20	0.053	6

   Table 6 shows the overall mean and standard deviation of the result.

Mean and STD of testing error	0.041 ± 0.004
Mean and STD of d*	5.15 ± 1.013

1.1.5 Discussion

Parameter Cross-validation
We have used a cross-validation technique to search an optimum hyper parameter. However, throughout the experiments, we found out that the optimum hyper parameter we found may not be the optimal in different epoch size. In other words, we did not consider epoch size when cross-validating over the number of possible parameters of kernels.

In such cases, where we have to find multiple optimal hyperparameters, it is good to use 'Grid Search' method. Even though it is computationally expensive as it exhaustively finds the best pair of hyperparameters, it can assure you that the algorithm have tried out every possible combination of the parameters, and the final result from the grid search is the optimal set of parameters for the model.

Comparing results from One Vs. Rest & One Vs. One

$d =$	Training error (%)	Testing error (%)
1	0.100 ± 0.021	0.114 ± 0.021
2	0.036 ± 0.007	0.057 ± 0.009
3	0.021 ± 0.004	0.048 ± 0.006
4	0.016 ± 0.003	0.043 ± 0.005
5	0.015 ± 0.007	0.043 ± 0.008
6	0.017 ± 0.024	0.046 ± 0.026
7	0.010 ± 0.001	0.039 ± 0.004

$d =$	Training error (%)	Testing error (%)
1	0.014 ± 0.003	0.058 ± 0.003
2	0.0003 ± 0.0004	0.035 ± 0.002
3	0.0002 ± 0.0002	0.032 ± 0.005
4	0.0001 ± 0.0001	0.0308 ± 0.004
5	0.0002 ± 0.0002	0.0336 ± 0.003
6	0.0004 ± 0.001	0.035 ± 0.004
7	0.0001 ± 0.0001	0.033 ± 0.003

Table 1: Caption to be added

In Table 1, the left table is the train/test errors from One Vs. Rest, while the right side of the table is that from One Vs. One perceptron.
Firstly, as can be seen in both of the table, training error rate is the biggest when $d = 1$. We can infer that the decision boundaries, when $d = 1$, is placed between classes, which make the model relatively underfitted, compared to when d is not set to 1.

Also, we can notice one more thing that the training errors tend to decrease as d value gets bigger, and stops decreasing when d is set around 5 and 6. This behaviour aligns with the result we've seen from experiment 2 and experiment 6. The optimal kernel parameter value was mostly around 5 and 6. This is not a coincidence.

The difference that we can found from the two table is that training errors are much smaller from 'One vs. One' model, and this is explainable. 'One vs. One' method creates much more classifiers than 'One vs. Rest', and it only deals with two classes, rather than the whole classes. Once model can only deal with two classes, the decision boundary will be clearer. This is why training a 'One vs. One' model can give us lower training error than training a 'One vs. Rest' model.
This is not just helpful in decreasing training error. We can also expect lower testing error from 'One vs. One' model. If the decision boundary was set well enough to classify two specific classes, than we can expect better performance in testing.

However, there are also downsides of using 'One vs. One' model.
First, it is computationaly costly, because we have to create much more binary classifiers than when using 'One vs. Rest'. Second, when the training size gets bigger, the model can easily overfit the data, causing bad performance when testing.

Comparing results of the Gaussian and Polynomial Kernel
First, we can compare the best parameter values for each kernel.
Table 2 shows the train/test error of 'One vs. Rest' perceptron with either polynomial kernel or gaussian kernel.

Considering kernels with k-class generalisation method, Table 2 shows the testing errors and optimal parameter values depending on the kernel method and k-class gerneralization method.

	Polynomial & 'One vs. Rest'	Polynomial & 'One vs. One'	Gaussian & 'One vs. Rest'
Test error (%)	0.036 ± 0.004	0.041 ± 0.004	0.03 ± 0.004
Optimal hyper-parameter	$d^* = 5.45 \pm 0.589$	$d^* = 5.15 \pm 1.013$	$c^* = 0.009 \pm 0.0008$

$d =$	Training error (%)	Testing error (%)
1	0.100 ± 0.021	0.114 ± 0.021
2	0.036 ± 0.007	0.057 ± 0.009
3	0.021 ± 0.004	0.048 ± 0.006
4	0.016 ± 0.003	0.043 ± 0.005
5	0.015 ± 0.007	0.043 ± 0.008
6	0.017 ± 0.024	0.046 ± 0.026
7	0.010 ± 0.001	0.039 ± 0.004

$c =$	Training error (%)	Testing error (%)
0.0005	0.09 ± 0.0071	0.100 ± 0.007
0.001	0.07 ± 0.02	0.09 ± 0.016
0.005	0.003 ± 0.002	0.037 ± 0.005
0.007	0.001 ± 0.001	0.029 ± 0.002
0.008	0.002 ± 0.001	0.032 ± 0.005
0.009	0.0005 ± 0.0003	0.032 ± 0.004

Table 2: Train/Test error from polynomial kernel (Left) and gaussian kernel (Right)

Confusion Matrix
From experiment 3, we have created a confusion matrix in Figure 2 that represents the error rates of combinations of predictions on testing data. This was done by using polynomial kernel with 'One vs. Rest' generalization method.
Say the 10 different classes we see on the y-axis of the confusion plot, i, and the classes in x-axis as j. Then, we read the confusion heatmap plot as: percentage error that model makes wrong prediction in class i to class j.
We can then see that class 3 can be relatively easily misclassified as class 5.

It is sensible, in that number 3 and 5 can look very similar if badly written.

Other combination of mis-classification that is likely to happen are:

- class 5 classified as class 3
- class 4 classified as class 9
- class 8 classified as class 3 (and so on)

Hardest digits
Figure 3 shows the most five hardest digits to predict.
However,by looking at 3(a), 3(b), and 3(c), we can suspect that those images are wrongly labelled. They look more like 1's. Also, 3(e) has strong noisy data that can make the digit look like 3, rather than 9. After plotting these digits, we could understand that it is not surprising that these were hard for models to predict.

1.1.6 Implementation

Below is the python code that implemented both 'One vs. One' and 'One vs. Rest' perceptron with kernelized matrix.

```
def _sign(val):
        ret = np.where(val <= 0.0, -1, 1)
        return ret

def fit(self, train_indices, X, y):
        self.train_indices = train_indices

        for i in range(X.shape[0]):
          cls_vals = np.zeros((self.n_classifiers,))
          kernel_values = self.kernel_matrix[self.train_indices, self.train_indices[i]]
          cls_vals += self.alphas.dot(kernel_values)

          y_hat = self._sign(cls_vals)
```

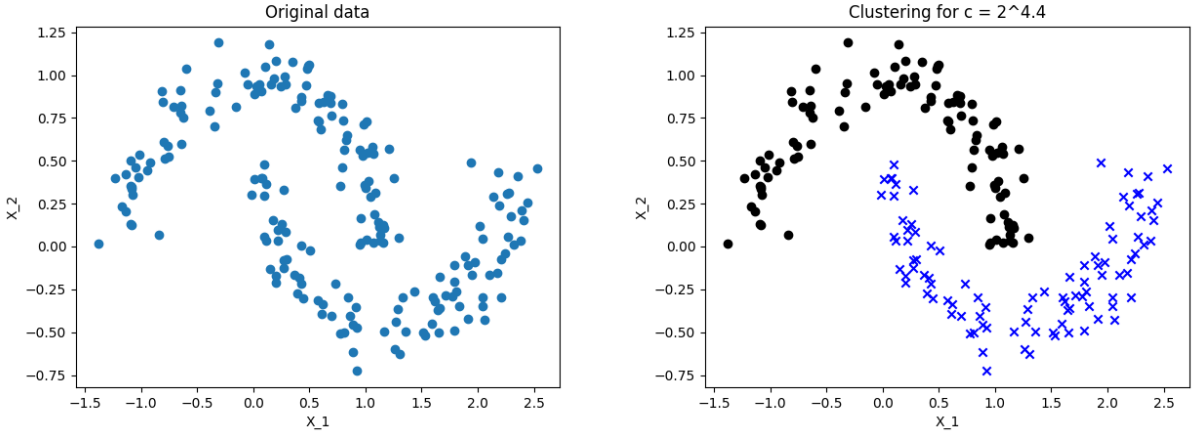
```
            y_true = self._encode_label(y[i])

        if self.method == 'ovr':
                condition = np.multiply(cls_vals, y_true) <= 0
                self.alphas[:, i] -= np.where(condition, y_hat, 0)
        elif self.method == 'ovo':
                condition = y_hat != y_true
                self.alphas[:, i] += np.where(condition, y_true, 0)

        return self.alphas

def predict_ovo(self, alphas, kernel_vals):
        """
        Only for prediction of one vs one
        """
        y_pred = (alphas.dot(kernel_vals) > 0) * 2 - 1
        vote = np.zeros((10, kernel_vals.shape[1]))
        for cls_i in range(self.classifier_values.shape[0]):
                row_pos = int(self.classifier_values[cls_i, 0])
                row_neg = int(self.classifier_values[cls_i, 1])

                pos = y_pred[cls_i, :] == 1
                neg = y_pred[cls_i, :] == -1

                vote[row_pos, pos] += 1
                vote[row_neg, neg] += 1

        return np.argmax(vote, axis=0)

def _encode_label(self, label):
        if self.method == 'ovr':
                labels = np.full((self.n_classifiers,), -1)
                labels[label] = 1
                return labels
        elif self.method == 'ovo':
                labels = np.zeros((self.n_classifiers,))
                for i in range(self.n_classifiers):
                        if self.classifier_values[i][0] == label:
                                labels[i] = 1
                        elif self.classifier_values[i][1] == label:
                                labels[i] = -1
                return labels
```

2 Part II

2.1 Spectral Clustering

2.1.1 Experiments

1. The plots for the original dataset and the clustered dataset can be seen in Figure 7. c = 4.4 is the value which correctly clusters the data.

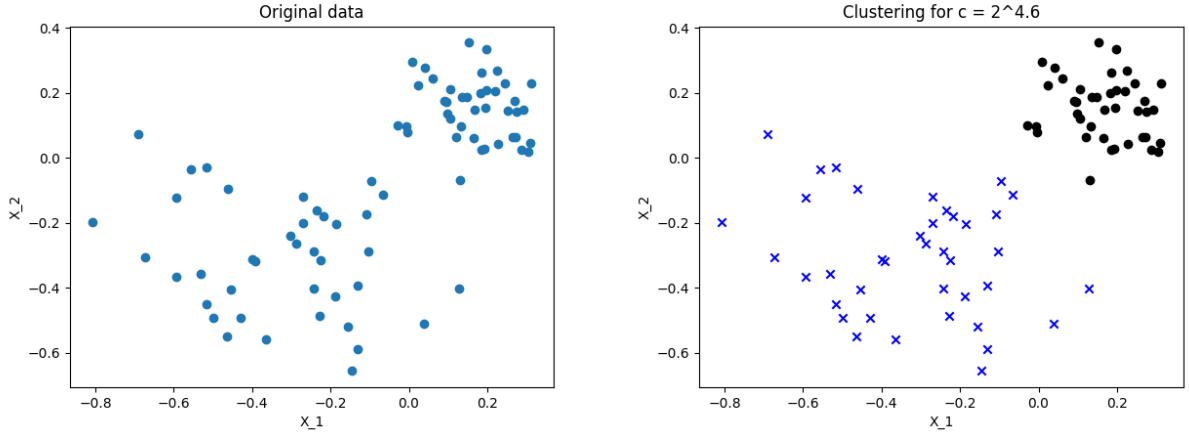


(a) original data

(b) clustered data for c = 4.4

Figure 7: Data before and after going through clustering algorithm

2. The plots for the original dataset and the clustered dataset can be seen in Figure 8. A value of c here that almost correctly clustered the data is 4.6, through testing this value varied a lot with the distribution, however, 4.6 worked best for the points shown.



(a) original data

(b) cluster with c = 4.6 producing 0.0125 clustering error

Figure 8: Data before and after going through Gaussian clustering algorithm

3. The plot comparing correctness against c can be seen in Figure 9. The value of c that worked best for this dataset was 0.04, producing a correctness of approximately 88.67%.

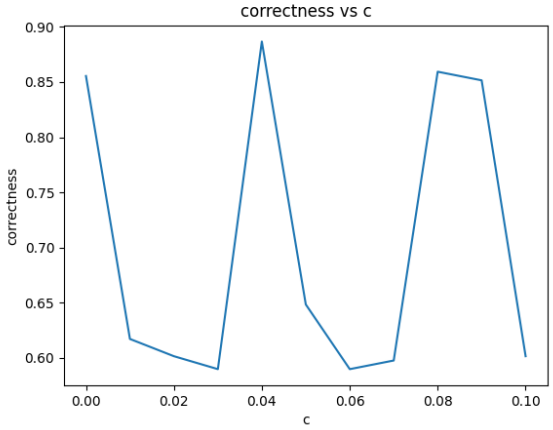


Figure 9: Plotting correctness against c, a greater correctness correlates to a better clustering

For my implementation of clustering, I firstly generated the array of c_values to test, and then pass those values one at a time to a clustering function along with the dataset. Within this function, I calculate the Weight matrix (W) as per the guidance in the brief, construct the Diagonal matrix (D) with it, and then the Laplacian matrix (L), followed by calculating the eigenvalues and eigenvectors on Laplacian with np.linalg.eig. Since np.linalg.eig does not guarantee that the eigenvalues would be in order, I use np.argpartition to put the 2^{nd} smallest eigenvalue in the 2^{nd} index, and consequently take the eigenvector at the corresponding index ($\mathbf{v}_2$). Once I have acquired $\mathbf{v}_2$, I can run the sign function on it to see if the values are above or below 0 and classify them accordingly. With this, I have a set of labels which I can iterate over to see how much error the clustering algorithm generated for the given value of c. After running the algorithm on all values of c, I can search for the one which most correctly clustered the data, and plot the clustered data it generates.

2.1.2 Questions

1. ℓ_+ is the number of correct classifications while ℓ_- is the number of incorrect classifications, since CP(c) takes the maximum of those 2 numbers, if ℓ_+ was large, it means that there were a large number of correct classifications, so $\frac{\ell_+}{\ell}$ would be close to 1 and represented greater cluster correctness. If ℓ_- was large, it means that there was a large amount of mis-classification, however, it would mean that the inputs have still been separated into clusters which can be inverted and will represent the same cluster correctness. Therefore, regardless of whether there were large correct or incorrect classifications, CP(c) is a reasonable measure of cluster correctness.

2. The first eigenvector is the all ones vector since the graph is connected, and the eigenvalue of the all ones eigenvector is 0.

3. Spectral clustering works as it is projecting the data from a higher dimensionality into a lower dimensional space (i.e. the eigenvector domain) [1]. In this lower dimensionality, the data is easily separable and no assumption needs to be made about the shape of the cluster, which is an advantage of spectral clustering over k-means clustering. This lower dimensional space has encoded information about the proximity between nearby data points [2] so any relationship between these points represents a cluster in original dataset.

   [1] https://www.cs.cmu.edu/ aarti/Class/10701/slides/Lecture21_2.pdf

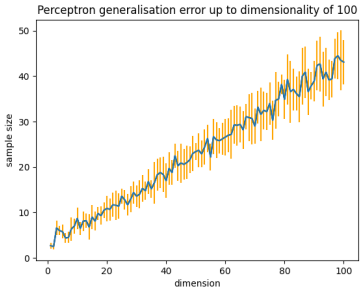
   [2] https://www.quora.com/How-does-spectral-clustering-work

4. C controls the size of the cluster being searched for, it does this by varying the weighting of proximity between points, a larger value of c will bias the weightings to form tighter clusters while smaller values of c will allow for larger clusters.

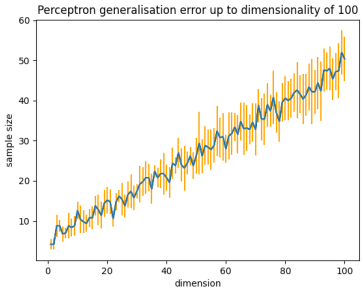
3 Part III

3.1 Questions

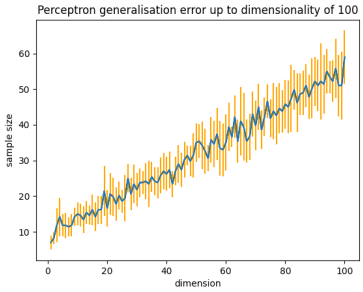
1. (a) Sample complexity plots for the Perceptron, Winnow, Least Squares, 1-NN, can be seen in figures 10, 11, 13 and 14 respectively



(a) increment=1

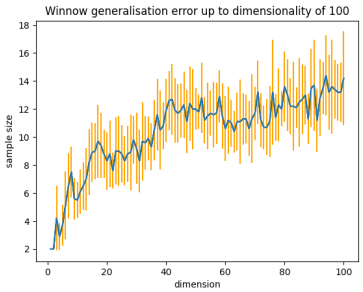


(b) increment=2

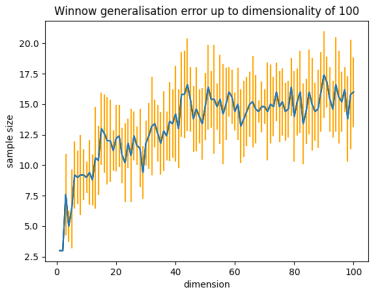


(c) increment=4

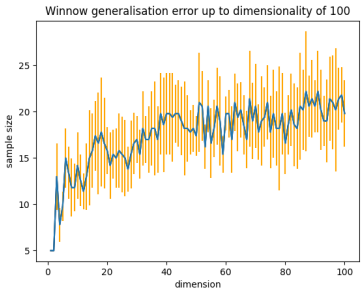
Figure 10: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for Perceptron for n=100, while incrementing the size of the training set by 1, 2 and 4



(a) increment=1



(b) increment=2



(c) increment=4

Figure 11: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for Winnow for n=100, with a testing sample size of 1000, while incrementing the size of training set by 1, 2, and 4

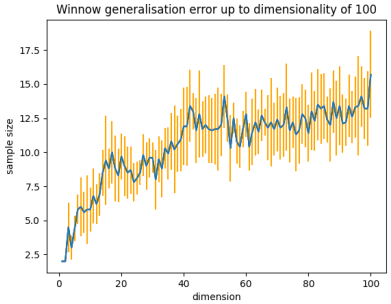
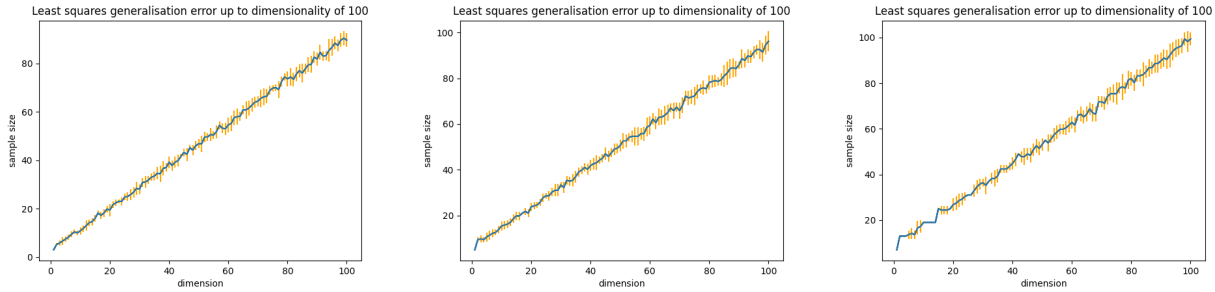
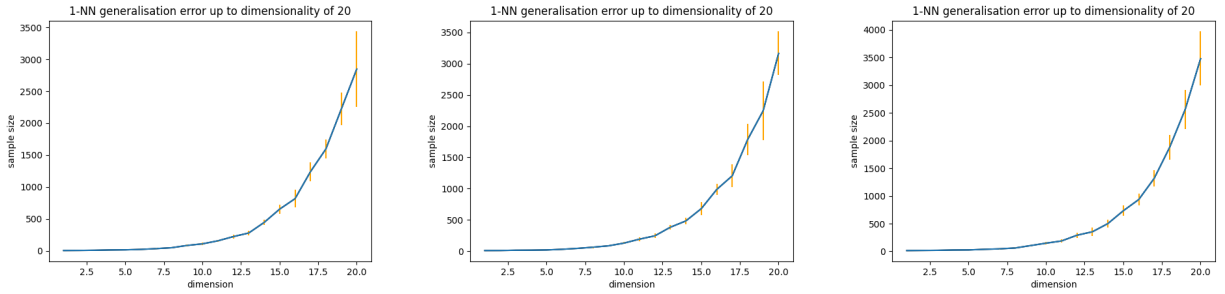


Figure 12: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for Winnow for n=100, with a testing sample size of 100,000, while incrementing the size of the training set by 4



(a) increment=2 (b) increment=4 (c) increment=6

Figure 13: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for LSE for n=100, while incrementing the size of the training set by 2, 4, and 6



(a) Incrementing in steps of 2 (b) Incrementing in steps of 4 (c) Incrementing in steps of 6

Figure 14: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for 1-NN for n=20, while incrementing the size of the training set by 2, 4 and 6

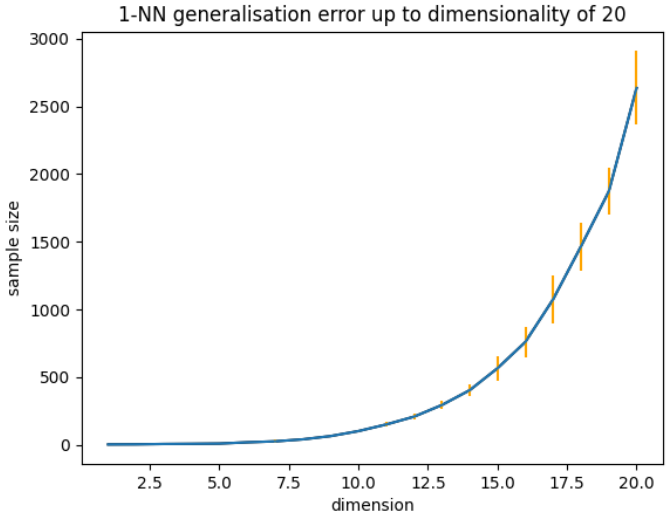


Figure 15: Estimated number of samples (m) to obtain 10% generalisation error versus dimension (n) for 1-NN for n=20, with a testing sample size of 1000, while incrementing the size of the training set by 1

(b) i. My method for estimating sample complexity for the perceptron, winnow and least squares were each very similar.

I had a function which could generate a training dataset (varied depending on the parameter passed to it) and testing dataset (fixed to 1,000 samples as it seemed reasonable for achieving a good estimate given the processing power available, I tested sample sizes of 100,000 for winnow but it only decreased the variance of the classifier and not a worthwhile amount to sacrifice computation time, can be seen in Figure 12, it was only computable in a somewhat reasonable time as I incremented the training dataset size by 4, which already increases variance a lot).

I had then had a function which would run the classifier on the training and testing datasets, involving calculating the weight vectors for with the training dataset, and then using it to predict the label for the samples in the training dataset. After the classifier returns the set of predictions, it could be compared against the correct labels for the training dataset, and a generalisation error could be calculated (the percentage of mistakes made when predicting on a testing dataset).

I used an if statement to check if the generalisation error was below 10%, since the initial training dataset sizes were small, the classifier wouldn't necessarily be trained enough to achieve fewer than 10% mistakes. In the case that the generalisation error was above 10%, I increment the sample size (m) for the training dataset used to train the classifier by 1 and re-train the classifier on the new training dataset. The same testing set was used for the next set of predictions to ensure that there was no variation due to using a different sample of points (as there were 2^n permutations of points). After finding the sample size required to reach less than 10% generalisation error, I ran the same loop again, 10 times in total (reasonable computation time), to find the average sample size required to reach less than 10% generalisation error. Doing this also allowed error bars to be generated to see how much variance there was for the selected parameters.

Once I had obtained the average sample size (m) required for 10% generalisation error, I incremented the dimensionality (n) and carried out the same function with the same parameters used for the testing dataset size and increment value for the training dataset. With all of these values I could plot my estimation for sample complexity, m versus n.

When estimating sample complexity for 1-NN, I discovered that the sample complexity was increasing exponentially and so had to vectorise many of the operations in order to run the algorithm in a reasonable time while not sacrificing accuracy too much. After this, I still needed to increment the training dataset size by greater values (2, 4 and 6) compared to 1, 2 and 4 used for the perceptron, winnow and least squares. I did run the algorithm once with an increment of 1 (Figure 15 and the variance was noticeably lower as dimensionality increased.

ii. The parameters which were adjusted and controlled were the following:
- dimensionality of the sample points
- the size of the training and testing datasets
- the steps size taken to increment the size of the training dataset in order to run the code in a reasonable time
- the number of times the experiment was carried out to get each average sample size (that resulted in 10% generalisation error)

The dimensionality was increased in order to plot the sample complexity, however, as this meant that there was a greater number of possible points that could be generated from the probability distribution, the size of the testing dataset would need to be proportionally increased in order to accurately calculate the true sample complexity. Not increasing the testing dataset during execution was a trade-off made to decrease accuracy for computation time. In fact, not testing on the full dataset that could be generated from the probability (size 2^n with n being the dimensionality) was a trade-off of accuracy for computation time. This is very apparent with

Figure 14 where the error bars increase as dimensionality increases.

There was no checking done to see if there were duplicate entries in the testing dataset, as for dimensionality smaller than 10 (for 1000 testing samples), there were bound to be duplicates in the data. This tradeoff was made so that the same testing set can be used for all predictions and saved computation time when estimating for values of n beyond 10 at the cost of accuracy in the generalisation error.

I increased the step size of the training dataset size increment as there wasn't a lot of value in testing every single size of training dataset other than to decrease variance, for the purposes of estimating sample complexity, I was obtaining good results even with larger step sizes of 6.

I calculated the sample size over an average of 10 runs which increased accuracy significantly in exchange for computation time. The average could have been computed over 5 runs but I was already sacrificing accuracy in other places and this increase did not add to computation time as much as others may have.

(c) Experimentally, from the plot given for the perceptron, it seems that sample complexity grows linearly as a function of dimension, it is $m = \mathcal{O}(n)$

Experimentally, from the plot given for winnow, it seems that sample complexity grows logarithmically as a function of dimension, it is $m = \Theta(log(n))$

Experimentally, from the plot given for least squares, it seems that the sample complexity grows linearly as a funciton of dimension, it is $\Theta(n)$

Experimentally and intuitively, from the plot given for 1-nn, it seems that the sample complexity grows exponentially as a function of dimension, it is $\Omega(n)$

(d) The probability of making a mistake on the s^{th} example after being trained on m examples with dimensionality n is $\hat{p}_{m,n}$.

As the sample complexity appears to be $\frac{n}{2}$ for generalisation error of 10%, this translates to a 10% chance of making a mistake after being trained on a sample size of m with dimensionality n. $\hat{p}_{50,100} = 0.1$ $\hat{p}_{\frac{n}{2},n} = 0.1$

With the dataset described where $\mathbf{x}_1, \cdots \mathbf{x}_m$ is sampled uniformly at random from $\{-1, 1\}^n$, the datapoints will be linearly separable with margin (γ) 1 for all n, since the closest 2 points in different classes can be is when the first x-coordinate is 1 and -1, with the rest of the points being the same, thus the equation for euclidean distance will be $\sqrt{(1+1)^2 + (1-1)^2 + (1-1)^2 + \cdots}$ which equates to 2, divided by 2 results in a margin of 1. All of the points are bound by a sphere of radius R, where R is $\sqrt{(1-0^2 + (1-0)^2 + \cdots}$ which is equivalent to $\sqrt{n}$. As $R = \sqrt{n}$ and $\gamma = 1$, the maximum number of mistakes that the perceptron algorithm can make is $\left(\frac{R}{\gamma}\right)^2$ which is n.

(e)