3 Kernel Perceptron

(1) Each of 20 runs used a randomly (and uniquely) split ‘zipcombo‘ dataset into 80% train and 20% test.
Each run for each polynomial degree d from 1 to 7 was performed on train and test datasets.
The mean train and test error rate percentages and standard deviations across the 20 runs are shown in Table 1 (to 2 d.p.):

d	**train error (%)**	**test error (%)**
1	7.58±0.84	1.84±0.22
2	1.35±0.47	0.85±0.13
3	0.47±0.01	0.69±0.11
4	0.25±0.15	0.63±0.10
5	0.13±0.05	0.60±0.09
6	0.09±0.04	0.60±0.06
7	0.07±0.03	0.62±0.07

Table 1. Mean train and test error rates (%) across 20 runs with polynomial kernel

(2) A“best” parameter d^* is determined 20 times by 5-fold cross-validation of the 80% training data split.
Using this d^*, we re-train weights with the full 80% training set and run tests on the remaining 20% errors.
Of the 20 d^*s, the mean and standard deviation of d^* was 5.55 ± 0.67. Of the same 20 for which tests were run (on the 20% dataset), the mean and standard deviation test error was 0.60±0.09%.

3 Kernel perceptron contin.

(3) The same operations as question 2 were repeated, replacing mistake counts with confusion matrices. The values in this matrix are calculated by counting the number of instances whereby a particular digit (‘true y’) was mis-classified (‘predicted y’), such that every combination of mis-classifications is added up and divided by the total count of that particular digit, giving the mean values (Figure 1).

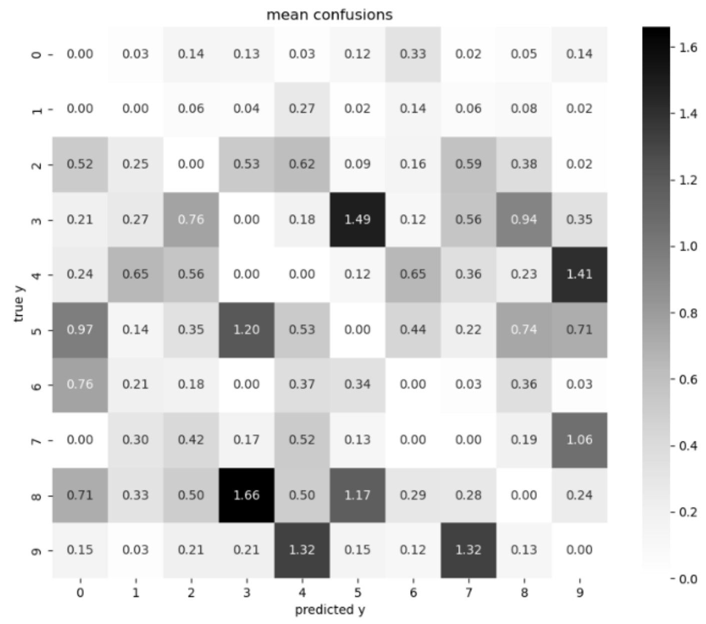


Figure 1. Mean confusion rates from 20 runs, on test datasets.

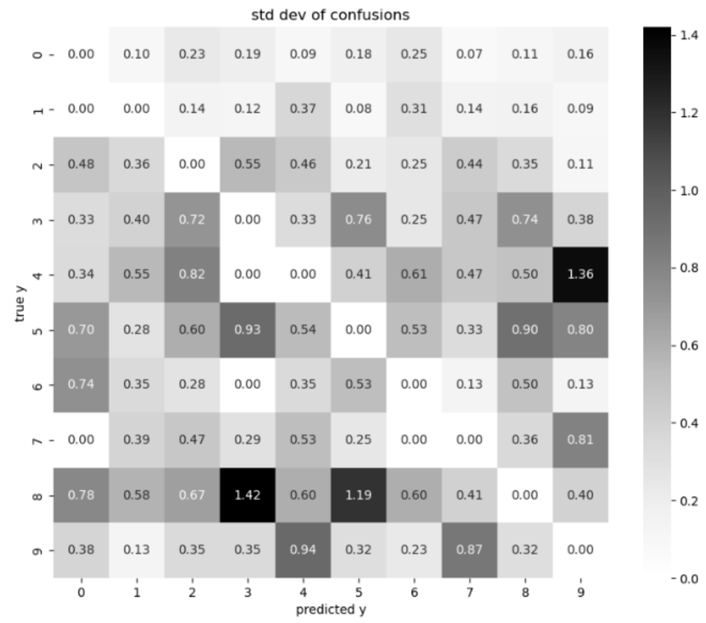


Figure 2. Standard deviations of confusion rates from 20 runs, on test datasets.

3 Kernel perceptron contin.

(4) The sum of confusion error rates for each digit, according to the 20% test set (and 3 epochs) is shown in Table 2 below:

0	1	2	3	4	5	6	7	8	9
0.99	0.69	3.16	4.88	4.22	5.30	2.28	2.79	5.68	3.64

Table 2. The sum of confusion error rates for each digit (20% test set)

So five hardest-to-predict numbers according to the confusion matrix are: 8, 5, 3, 4, 9 (from the hardest).

The wording of the question suggests that it may be after individual images (rather classes), and potentially from the full dataset, though it's not completely clear. Nonetheless, weights were trained on the full data set (3 epochs), degree 5 (rounded down from mean d^* in Q2. ($d = 5.55$ was tried but it caused numerical instability issues). Predictions were then made, again on the full dataset, using the trained weights. 19 digits were mis-classified out the 9298 digits that are in the full dataset, shown in Figure 3 below:

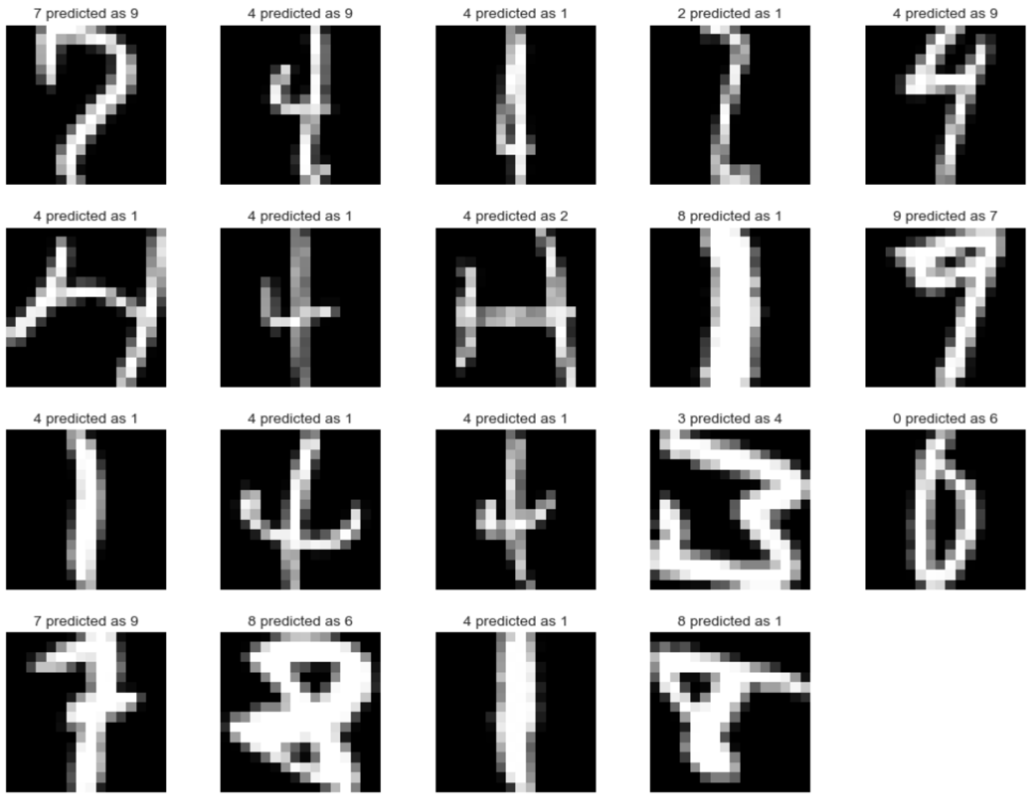


Figure 3. 19 incorrectly predicted images from the full dataset of 9298 digits, using polynomial kernel.

3 Kernel perceptron contin.

(4) contin.
Some of these mis-classifications are more surprising than others.
The high prevalence of 4s, numbering at 10 of the 19, is not surprising given how varied their appearances are in this set.
The 7s and 9s are confused twice, which is not so surprising.
A 6 and 8 confused is not all that surprising.
There are several examples where a number other than 1 looks like a 1, and so it is not surprising when these are 'wrongly' predicted as 1s.
The 8 in the row#4, column#4 is predicted to be a 1 though it looks more like either a 9, 4 or an 8.
Predictions for row#2, column#1 and row#2, column#3 are more surprising being predicted as a 1 and 2.
Regardless, it seems remarkable that only 19 of 9298 were mis-classified given such a simple algorithm - there are no convolutional neural networks here, just a 'kernel trick' and a basic perceptron.
Perhaps a more scientific treatment of "surprise" is to note that there are cases where the data is just poor. A model that generalises to classify new examples correctly should **not** be able to correctly classify a number 8 that a human would not recognise as a number 8 - such as in row#2, column#4. This was classified as a 1, and could in fact be considered as a correct classification or a 'false' mis-classification. This is repeated for digits in row#3, column#1 and row#4, column#3, where there are 4s that a human would also identify as a 1. In these cases, it is correct to say that the kernel perceptron has successfully classified the image, despite it disagreeing with the true label. At the same, there are cases where the model can be considered to have failed and suggests therefore that it has room for improvement. If one is restricted to this dataset only, improvement might be achievable by use of more epochs, a learning rate, and/or regularisation. The higher test error rates (in Table 1) suggest that use of a learning rate and regularisation could reduce some overfitting, which potentially could result in fewer 'true' mis-classifications.

Table 3 below indicates the percentage rate of mis-classification per digit.
An explanation of the column names:
'total' is the sum of occurences of each digit in the full dataset.
'mis-classified' is the sum of mis-classified occurences (these are the 19 mis-classified digits shown in Figure 3 above):
'rate (%)' is the mis-classified divided by the total, multiplied by 100.

digit	**total**	**mis-classified**	**rate** (%)
0	1553	1	0.06
1	1269	0	0
2	929	1	0.11
3	824	1	0.12
4	852	10	1.17
5	716	0	0
6	834	0	0
7	792	2	0.25
8	708	3	0.42
9	821	1	0.12

Table 3. Polynomial kernel rates of mis-classifications with full dataset (%)

(5) Repeating 1 and 2 with a Gaussian kernel:
The approach taken for selecting a range of c hyperparameter was entirely empirical. Starting first with very small subsets of the dataset, a wide range of c was tried out. The initial guess of the range of values to try was very close to that eventually used with 10-fold differences from 0.0001 to 50. This seemingly crude approach yielded very low test errors across a broad range of c values between 0.001 and 1, with the lowest at 0.01 (0.59±0.08%).

c	**train error (%)**	**test error (%)**
0.0001	15.77±2.83	3.21±0.49
0.001	6.15±1.68	1.53±0.39
0.01	0.17±0.06	0.59±0.08
0.1	0.07±0.07	1.07±0.12
1	0.03±0.02	1.36±0.11
10	0.00±0.01	3.91±0.14
50	0.00±0.00	13.41±0.2

Table 4. Mean train and test error rates (%) across 20 runs with Gaussian kernel

A "best" parameter c^* is determined 20 times by 5-fold cross-validation of the 80% training data split.
Using this c^*, we re-train weights with the full 80% training set and run tests on the remaining 20% errors.
Of the 20 c^*s, the mean and standard deviation was of c^* 0.01 ± 1.73. (This high standard deviation is not surprising given the 10-fold difference between the different c values used). Of the same 20 for which tests were run (on the 20% dataset), the mean and standard deviation test error was 0.59±0.08%.

3 Kernel perceptron contin.

(6) One possible alternative method might be to combine the polynomial and Gaussian kernel perceptrons.
The digits that were misclassified by the polynomial kernel were highlighted in the confusion matrix (Figure 1) and 19 misclassified in full dataset (Figure 3). This was repeated with the Gaussian kernel in order to see certain each kernel is better/worse at classifying certain digits compared to each other. The idea being that one might then apply a weighted combination of their predictions according to their relative strengths/weaknesses with regards to individual classes.
Indeed, it appears, from the full dataset, that the Gaussian kernel (using c=0.01) may be worse at predicting 8s than the polynomial kernel (d=5) which may be worse at predicting 4s (Figure 4).

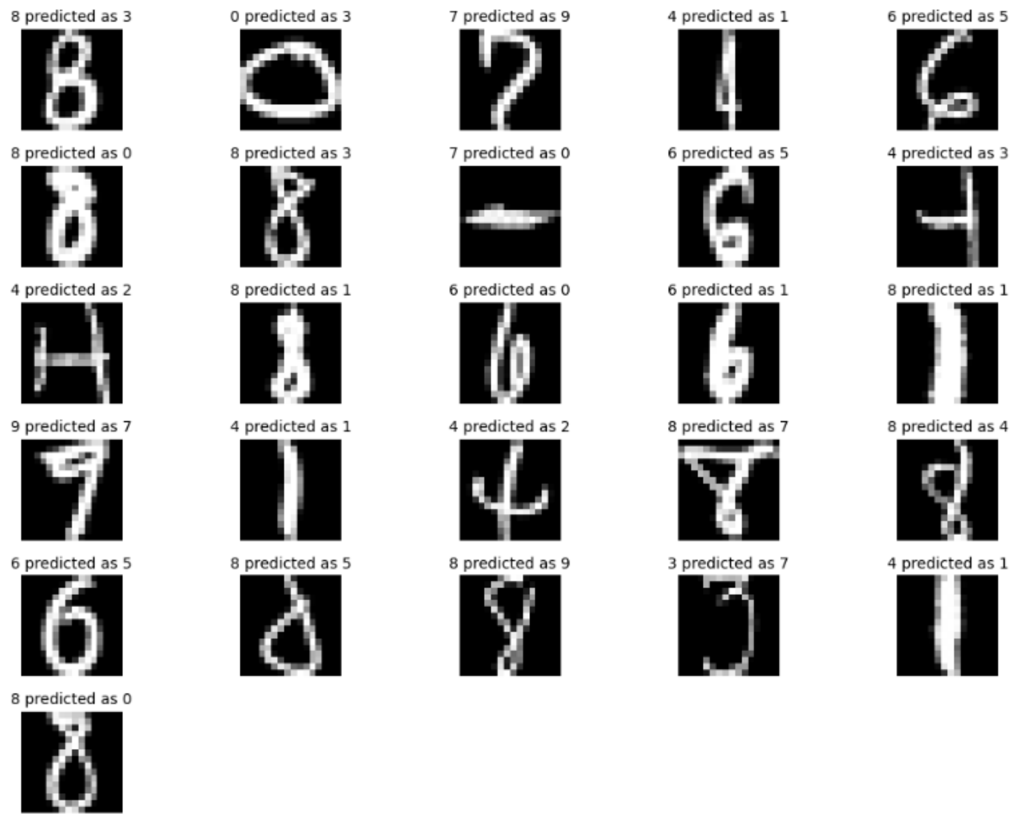
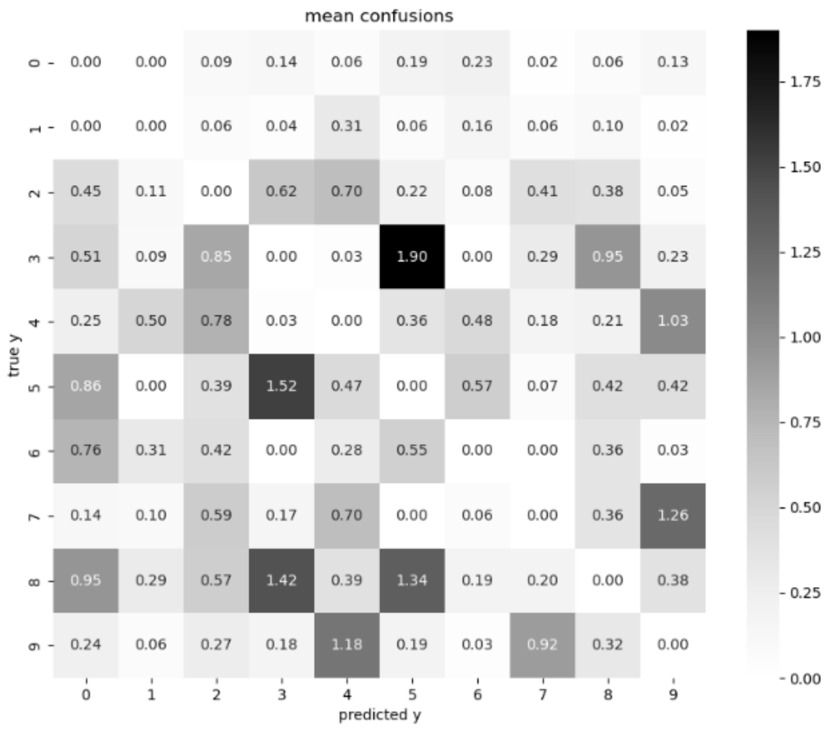


Figure 4. 26 incorrectly predicted images from the full dataset of 9298 digits, using Gaussian kernel.

As such, the updates would make use of both kernel simultaneously, applying perhaps a 50-50 weighting for most of the 10 classifiers, but a weighted preference for the polynomial's 8-classifier, while applying a weighted preference for the Gaussian's 4-classifier. The weights may be set according to the actual numbers of mis-classifications/confusions.

3 Kernel perceptron contin.

(6) Contin.
On the whole though, with test data, the confusion rates are lower using a Gaussian kernel. Figure 5 shows the mean confusion rates from 20 runs, with values multiplied by 100 just to make the rate numbers visible (otherwise most of the matrix would display "0.00").



true y \ predicted y	0	1	2	3	4	5	6	7	8	9
0	0.00	0.00	0.09	0.14	0.06	0.19	0.23	0.02	0.06	0.13
1	0.00	0.00	0.06	0.04	0.31	0.06	0.16	0.06	0.10	0.02
2	0.45	0.11	0.00	0.62	0.70	0.22	0.08	0.41	0.38	0.05
3	0.51	0.09	0.85	0.00	0.03	1.90	0.00	0.29	0.95	0.23
4	0.25	0.50	0.78	0.03	0.00	0.36	0.48	0.18	0.21	1.03
5	0.86	0.00	0.39	1.52	0.47	0.00	0.57	0.07	0.42	0.42
6	0.76	0.31	0.42	0.00	0.28	0.55	0.00	0.00	0.36	0.03
7	0.14	0.10	0.59	0.17	0.70	0.00	0.06	0.00	0.36	1.26
8	0.95	0.29	0.57	1.42	0.39	1.34	0.19	0.20	0.00	0.38
9	0.24	0.06	0.27	0.18	1.18	0.19	0.03	0.92	0.32	0.00

Figure 4. Mean confusion rates from 20 runs, on test datasets, using Gaussian kernel.

A.
Only degrees d or width c were cross-validated over. Other parameters/hyperparameters that might have been cross-valdiated over include epoch number, numbers of runs, learning rate, regularisation. The explicit instructions in the questions were prioritised as much as possible, but the aforementioned parameters/hyperparameters would have been cross-validated over with more time and/or available compute power.

B.
The initial method chosen for generalised to a k-class classifier was one-vs-all. This was the interpretation of the method used in the mathematica code, wherein all 3 classifiers are trained and the one with the highest prediction score out of the 3 is the prediction.

C.

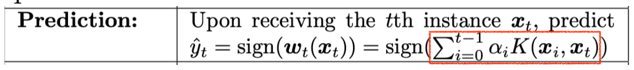
Comparison of results of the Gaussian to the polynomial Kernel. It was expected that the results would be noticeably better with the Gaussian kernel than the polynomial. For one, it is known that with a polynomial kernel, your data is not guaranteed to be separable, whereupon the kernel perceptron will not likely converge, whereas with a Gaussian kernel you are guaranteed to converge. However for the given dataset, and over the limited extent of parameter experimentation, the test errors with an optimal degree and width was remarkably low for both polynomial and Gaussian kernels, at 0.6±0.09% and 0.59±0.08%, respectively.

D.

Our implementation of the kernel perceptron follows the description in the question sheet and mathematica code:
Epochs were modeled by duplication of the dataset (which was implemented with NumPy's tile() function), such that epoch=3 resulted in the dataset tripled in size, the dataset repeating 3 times in one array:

$$\underbrace{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_{40}, y_{40})}_{\text{epoch 1}}, \underbrace{(\boldsymbol{x}_{41}, y_{41}), \ldots, (\boldsymbol{x}_{80}, y_{80})}_{\text{epoch 2}}, \ldots, \underbrace{(\boldsymbol{x}_{(m-1)\times 40+1}, y_{(m-1)\times 40+1}), \ldots, (\boldsymbol{x}_{(m-1)\times 40+40}, y_{(m-1)\times 40+40})}_{\text{epoch m}}$$

To implement the kernel perceptron, a 'pseudo-online learning' method was used, whereby the kernel matrix is pre-computed, but updates are performed one data point at a time. For each data point (aka time point or step), the dot product is taken of a slice of the kernel matrix and the correspondingly shape-matched slice of the weights array (α). This is done by iterating over the datapoints in the epoch-duplicated dataset in a for-loop. The slices of α and the kernel matrix are taken from the beginning (i.e. 0-index) up to the current iteration, such that the dot product serves to compute the sum of all "previous" multiplications of α and the kernel matrix. This is in line with the description in the prediction section of the question sheet:



Prediction:	Upon receiving the tth instance $\boldsymbol{x}_t$, predict $\hat{y}_t = \text{sign}(\boldsymbol{w}_t(\boldsymbol{x}_t)) = \text{sign}(\boxed{\sum_{i=0}^{t-1} \alpha_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)})$

However, the signum is applied to the true y labels, not to the dot product of α and the kernel evaluation - as shown in the equation from the question sheet, but rather as it is in the mathematica code.
Following the mathematica code further, the predicted value, $\hat{y}$, is not compared with the true label, y, but is multiplied by the prediction. A correct prediction is inferred if the product gives a positive (scalar) value, otherwise it is deemed to be incorrect such that the weights should be updated. The update is implemented by directly assigning the value of the true label y to the weights for this data point to each of the 10 classifiers. However, α is not changed at all and not assigned to 0 if the prediction is deemed to be correct, which seems not to be the update graphic on the question sheet.

Update:

```
if $\hat{y}_t = y_t$ then $\alpha_t = 0$
             else $\boxed{\alpha_t = y_t}$
$\mathbf{w}_{t+1}(\cdot) = \mathbf{w}_t(\cdot) + \alpha_t K(\boldsymbol{x}_t, \cdot)$
```

The update also appears slightly different in the mathematica code in terms of updating the weights by the value of the prediction, (via the signum function).

```
(* update *)
If[y preds⟦j⟧ ≤ 0,
 GLBcls⟦j, i⟧ = GLBcls⟦j, i⟧ - mysign[preds⟦j⟧]] ;
```

As such, our implementation takes features of each of these two forms, which may or may not be exactly the way it is being instructed to be implemented, but produces a very effective result. Three epochs were chosen and were not compared with 1, 2, 4, 5, or anything other, so it's quite possible that the accuracy of the predictions could be further improved. This is implemented in 'shared_functions.train_kp() function'. (**w** is not represented or directly calculated. All the learning is done by updates to the variable called α, which contains the coefficients of the classifiers).

E.

(Tables are described where they appear.)