

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	test
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.

- (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 was 5.55 ± 0.67 . Of the same 20 for which tests were run (on the 20% dataset), the mean and standard deviation was 0.60 ± 0.09 .

3 Kernel perceptron contin.

- (3) The same operations as question 2 were repeated, replacing mistake counts with confusion matrices has been generated.

The values in this matrix are calculated by counting the number of instances whereby a particular digit (according to the true label y) was misclassified as another digit (the predicted label y), such that every combination of misclassifications is added up and divided by the total count of that particular digit, giving the mean values (Figure 1).

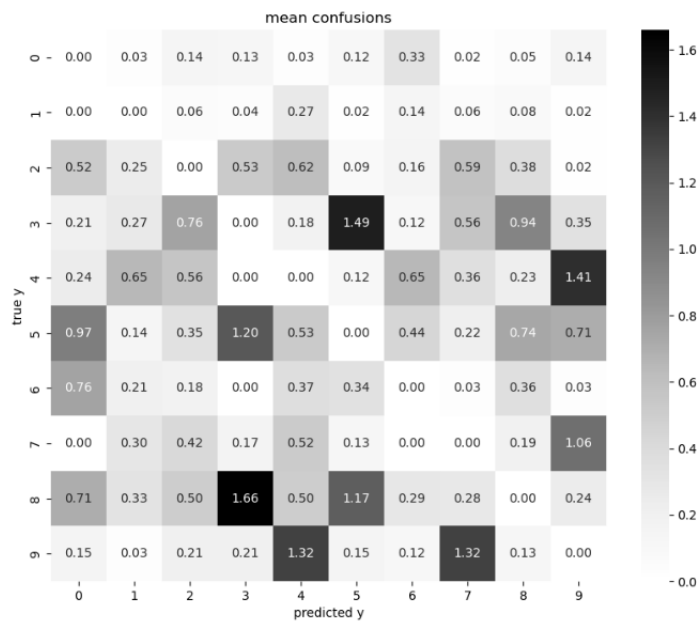


Figure 1. Mean confusions

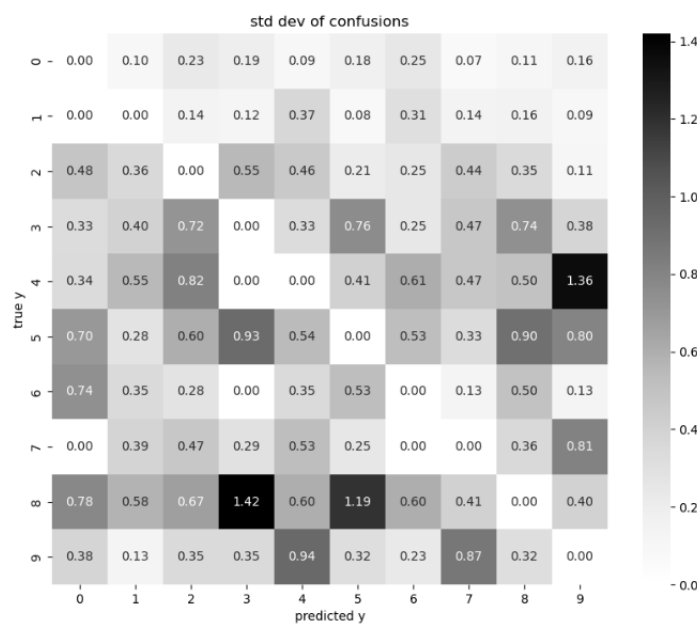


Figure 2. Standard deviations of confusions

- (4) The visualisation of the five hardest-to-predict digits along with their labels is shown in Figure 2 below:

Figure 3.

Yes/no, it is (not) surprising that these five are hard to predict, because....

3 Kernel perceptron contin.

- (5) Repeating 1 and 2 with a Gaussian kernel (d^* is now c and 1,...,7 is now S):
From 20 runs for $c = 1, \dots, 7$.

Basic results:

Each run uses a randomly split ‘zipcombo’ into 80% train and 20% test.

The mean train and test error rates and standard deviations are shown in Table 2 below:

d	train	test
1	00.00±0.0	00.00±0.0
2	00.00±0.0	00.00±0.0
3	00.00±0.0	00.00±0.0
4	00.00±0.0	00.00±0.0
5	00.00±0.0	00.00±0.0
6	00.00±0.0	00.00±0.0
7	00.00±0.0	00.00±0.0

Table 2. Error rates (%)

Cross-validation:

A “best” parameter c^* is determined by 5-fold cross-validation of the 80% training data split.

Using this c^* , we retrain on the full 80% training set. The test errors are computed on the remaining 20% using the same c^* .

The test error for this value of c^* is recorded and the process above is repeated 19 more times, resulting in 20 c^* and 20 test errors.

The mean test error and mean c^* is 00.00 ± 0.0 and 0.0 respectively.

3 Kernel perceptron contin.

- (6) An alternate method to generalise the kernel perceptron to k -classes might be to use ...:

Basic results:

Each run uses a randomly split ‘zipcombo’ into 80% train and 20% test.

The mean train and test error rates and standard deviations are shown in Table 3 below:

d	train	test
1	00.00±0.0	00.00±0.0
2	00.00±0.0	00.00±0.0
3	00.00±0.0	00.00±0.0
4	00.00±0.0	00.00±0.0
5	00.00±0.0	00.00±0.0
6	00.00±0.0	00.00±0.0
7	00.00±0.0	00.00±0.0

Table 3. Error rates (%)

Cross-validation:

A “best” parameter c^* is determined by 5-fold cross-validation of the 80% training data split.

Using this c^* , we retrain on the full 80% training set. The test errors are computed on the remaining 20% using the same c^* .

The test error for this value of c^* is recorded and the process above is repeated 19 more times, resulting in 20 c^* and 20 test errors.

The mean test error and mean c^* is 00.00 ± 0.0 and 0.0 respectively.

3 Kernel perceptron contin.

(6) **A.**

A discussion of any parameters of your method which were not cross-validated over.

B.

A discussion of the two methods chosen for generalising 2-class classifiers to k-class classifiers.

C.

A discussion comparing results of the Gaussian to the polynomial Kernel.

D.

A discussion of our implementation of the kernel perceptron:

$$\mathbf{w}(\cdot) = \sum_{i=0}^m \alpha_i K(\mathbf{x}_i, \cdot)$$

represents a function \mathbf{w} which takes some input (which here is expected to be the flattened array of MNIST image pixels for a digit and computes the weighted sum of kernel evaluations for this input and the image pixels for all the other digits in some dataset.

Alpha, the classifier, is an array of coefficients which are learned during training, according to mistakes made. The original mathematica code uses variable 'GLBcls' for alpha. We replaced this with 'alphas'.

The result of the weighted sum

(i) was represented by...

(ii) was evaluated by ...

(iii) had new terms added to it during training by ...

E.

(Any table produced in 1-6 above should also have at least one sentence discussing the table.)