COMP0078 Supervised Learning
Coursework 2

Daniel May
ucabdd3@ucl.ac.uk

November 26, 2020

Contents

1 Part I: Handwritten digit classification

1 Introduction

The MNIST dataset of handwritten digit images was introduced to the machine learning community in a 1998 paper [5], and has been widely used to train, test, and compare the performance of algorithms since.

Our variant of the dataset consists 9,298 handwritten digits, normalized to fit within $16x16$ grayscale pixels. An example of each class in the dataset is shown in Figure 1. We are tasked with comparing the properties and performances of various machine learning algorithms on classifying such images.

Figure 1: An example of each class in the dataset

In Figure 2, we plot a chart of the number of examples of each digit (0 to 9) that our dataset contains. There is a visible imbalance, as the number of examples the most frequent digit, 0 (1,553 examples), is almost double that of many other classes, with the least frequent being digit 8 (708 examples).

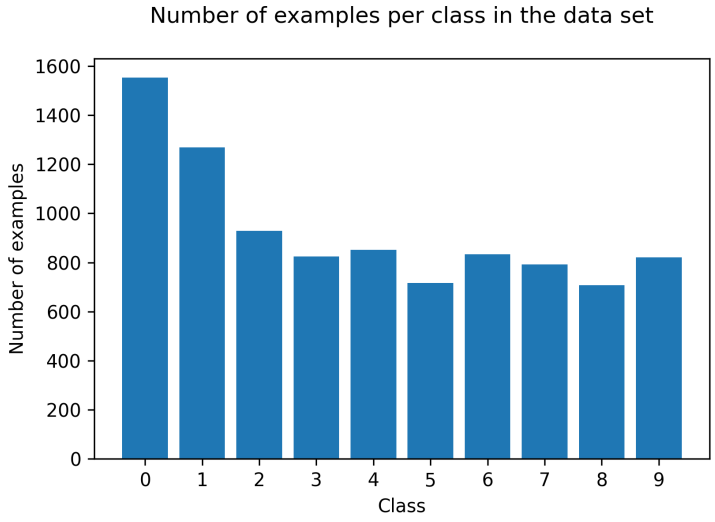
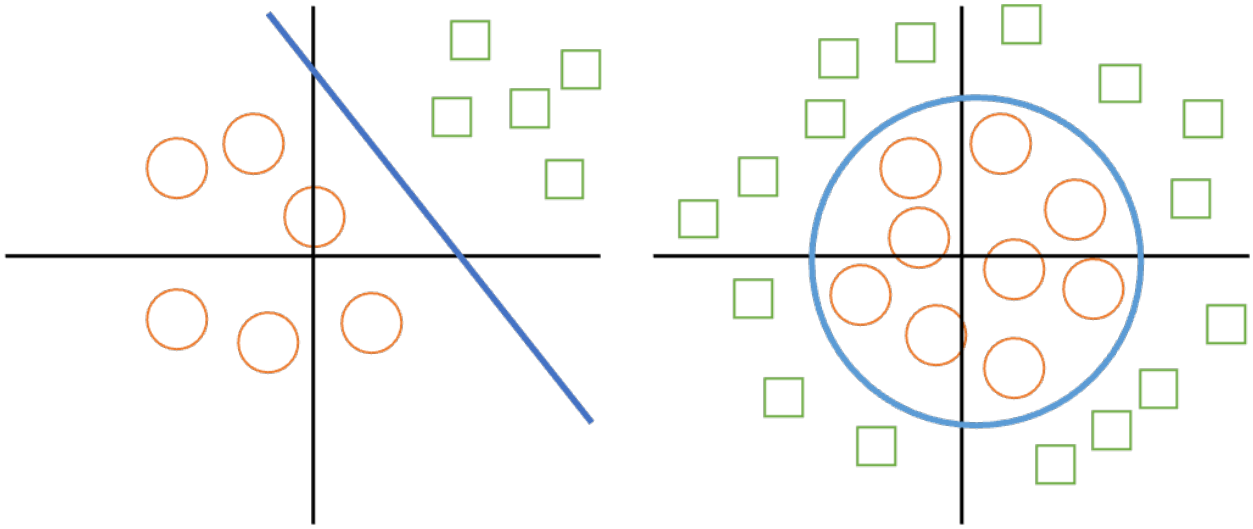


Figure 2: Number of examples per class in the dataset



(a) A linearly separable dataset; we can find a straight line that separates the two classes

(b) A non-linearly separable dataset; no straight line would separate the two classes. Ideally, we would learn a non-linear decision boundary, like the circular boundary displayed

Figure 3

(i) From perceptron to kernel perceptron

We will begin by investigating kernel perceptrons. The standard perceptron is a binary linear classifier, which iteratively learns a decision boundary, represented as a weight vector, which is a linear combination of the examples that the algorithm has misclassified so far. If the data are linearly separable (Figure 3a), Novikoff's theorem has shown that the algorithm is guaranteed to converge to a state where it can perfectly separate the data within an upper bound of updates.

The standard perceptron is shown in Algorithm 1. The algorithm is *online*, meaning it can operate on each example as it arrives. The *for* loop can be repeated for multiple epochs, and until a convergence criterion is met, such as that the training error rate is below a certain threshold. After training, the algorithm can be tested by carrying out the prediction step.

Algorithm 1: Perceptron

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$

1. Initialize $\boldsymbol{w}_1 = \boldsymbol{0}$
2. **for** *t=1 to m* **do**
   - Receive pattern: $\boldsymbol{x}_t \in R^n$.
   - Predict: $\hat{y}_t = sign(\boldsymbol{w}_t \cdot \boldsymbol{x}_t)$.
   - Receive label: y_t
   - **if** $\hat{y}_t y_t \leq 0$ **then**
     - $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + y_t \boldsymbol{x}_t$
   - **else**
     - $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t$
   - **end**

end

We call this the primal form of the perceptron, as it keeps track of its weight vector explicitly as it sees each example. By contrast, since the weight vector is simply a linear combination of the misclassified examples, their labels, and number of updates, we can represent it implicitly in a dual form by storing these.

In other words, the weight vector can be written as the sum

$$\boldsymbol{w}_m = \sum_{t=1}^{m} \boldsymbol{\alpha}_t y_t \boldsymbol{x}_t$$

where $\boldsymbol{\alpha}_t$ tracks the number of mistakes on $\boldsymbol{x}_t$. We can then make a prediction for $\boldsymbol{x}$ using

$$\hat{y}_t = sign(\boldsymbol{w}_t \cdot \boldsymbol{x}) = sign(\sum_{t=1}^{m} \boldsymbol{\alpha}_t y_t \boldsymbol{x}_t)^T \boldsymbol{x} = sign \sum_{t=1}^{m} \boldsymbol{\alpha}_t y_t (\boldsymbol{x}_t \cdot \boldsymbol{x})$$

which no longer involves the weight vector.

We can swap the dot products $\boldsymbol{x}_t \cdot \boldsymbol{x}$ for a kernel function, which computes a dot product as if the examples were mapped into a high-dimensional, non-linear feature space.

For example, we could choose a polynomial kernel function $K_d(\boldsymbol{x}_t, \boldsymbol{x}) = \phi(\boldsymbol{x}_t)^T \phi(\boldsymbol{x}) = (\boldsymbol{x}_t \cdot \boldsymbol{x})^d$, where $\phi(\cdot)$ is a feature map from an example to a space containing every possible order d polynomial term. As the size of $\phi(\cdot)$ grows exponentially in d, computing the product of feature maps may be intractable, requiring $O(m^d)$ computations, but kernel methods allow you to compute an equivalent, simpler function, like $(\boldsymbol{x}_t \cdot \boldsymbol{x})^d$, which is only linear in the number of examples, or $O(m)$.

This may be feasible as we are using the dual form; in the primal form, the weight vector would be a linear combination of high-dimensional feature mappings, which may be too large to represent explicitly. The dual representation is more efficient if the number of examples is significantly smaller than the size of the feature map.

With this set up, we can construct a kernel perceptron algorithm, which is again a binary classifier, but with the ability to learn a non-linear decision boundary (Figure 3b). Instead of storing a weight vector, we store all of the training examples and a vector $\boldsymbol{\alpha}$, which counts the number of times each example has been misclassified. The algorithm is shown in Algorithm 2. Again, the *for* loop can be repeated until a convergence criterion is met.

Algorithm 2: Kernel perceptron

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$

1. Initialize $\boldsymbol{\alpha}_1 = \mathbf{0}$
2. **for** *t=1 to m* **do**
   - Receive pattern: $\boldsymbol{x}_t \in R^n$.
   - Predict: $\hat{y}_t = sign \sum_{i=1}^{m} \boldsymbol{\alpha}_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)$
   - Receive label: y_t
   - **if** $\hat{y}_t y_t \leq 0$ **then**
     - $\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_t + y_t$

end

The elements of the $\boldsymbol{\alpha}$ vector are initialized to zero, so the computation in the *predict* step starts off trivial. However, as each mistake is made, a coefficient of $\boldsymbol{\alpha}$ is incremented or decremented, increasing the number of non-zero elements in $\boldsymbol{\alpha}$. As can be seen from the *predict* step, each additional non-zero element of $\boldsymbol{\alpha}$ requires the evaluation of an additional kernel function, so prediction may become increasingly costly. It may therefore be desirable to concern ourselves with the number of non-zero $\boldsymbol{\alpha}$ elements, and we will compare the numbers for classifiers with polynomial and Gaussian kernel functions.

(ii) Generalizing to k classes

A non-linear classifier like the kernel perceptron in Algorithm 2 is a good starting point for our handwritten digit recognition task. However, our dataset comprises ten classes (digits 0 to 9), so we need to generalize the algorithm to a multi-class setting. We investigate *one-versus-rest* and *one-versus-one* approaches and their limitations, then briefly discuss alternatives.

One-versus-rest In the one-versus-rest approach, a classifier is trained on each class, with examples from the corresponding class considered positive (+1) and all other classes negative (-1). For a k-class problem, this requires training k classifiers, so the approach is linear, or $O(k)$, in the number of classifiers. An example is shown in Figure 4.

Each classifier now outputs a *confidence* value rather than a sign. For the kernel perceptron, this means outputting $\sum_{i=1}^{m} \boldsymbol{\alpha}_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)$ itself rather than the sign of this value. The overall prediction is the class of the classifier with the greatest confidence that the example belongs to its class.

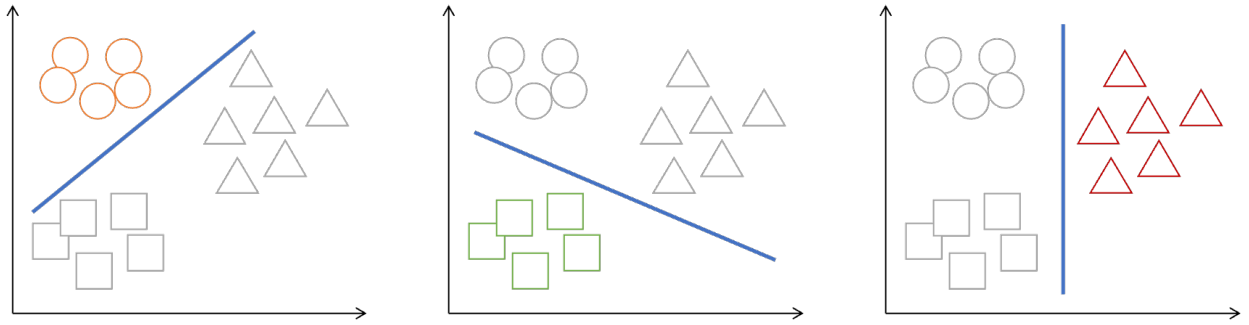


Figure 4: An example of the one-versus-rest approach on a 3-class dataset. A classifier is trained on each class, where examples from the corresponding classes are considered positive (+1) and all other classes negative (-1)

One-versus-one In the one-versus-one approach, a classifier is trained on each pair of classes, with examples from one class are considered positive (+1) and the other negative (-1). For a k-class problem, this requires training $k(k-1)/2$ classifiers, so the approach is polynomial, or $O(k^2)$, in the number of classifiers. An example is shown in Figure 5.

As in the binary setting, each classifier outputs a sign corresponding to the positive or negative class. For the kernel perceptron, this is $sign \sum_{i=1}^{m} \boldsymbol{\alpha}_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)$. For prediction, each classifier makes a prediction, and the overall prediction is the class which receives the greatest number of votes.

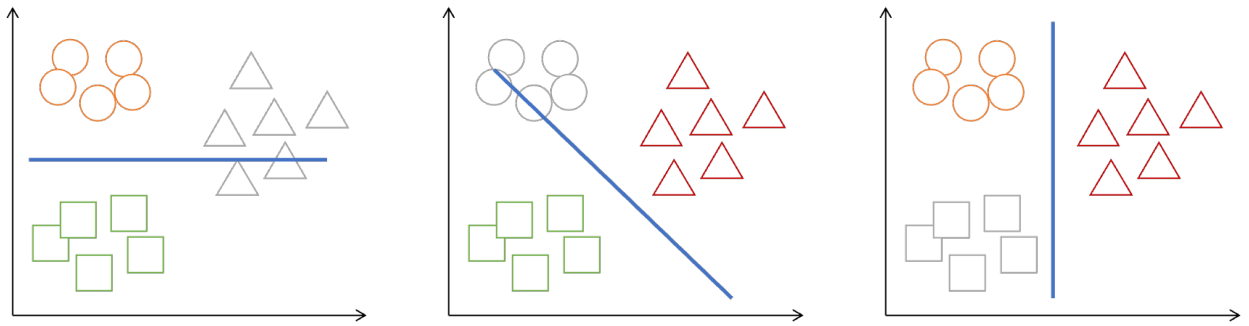


Figure 5: An example of the one-versus-one approach on a 3-class dataset. A classifier is trained for pair of classes, where examples from one class are considered positive (+1) and the other negative (-1)

Comparison As mentioned, one-versus-rest trains $O(k)$ classifiers for a k-class problem, while one-versus-all trains $O(k^2)$. This does not necessarily mean that one-versus-rest is more time efficient, as each classifier in one-versus-all will train on much less data, so each is significantly smaller. For example, an SVM is polynomial time in the number of examples to train, so it may make sense to use one-versus-all, despite the greater number of classifiers. We will investigate a one-versus-rest kernel SVM later.

Limitations and alternatives Both approaches suffer from a number of drawbacks.

In *one-versus-rest*, the classifiers will be greatly class imbalanced due to each classifier treating one class as positive and the remaining *k-1* as negative. Additionally, the *confidence* outputs of each classifier will typically have different scales, so the *argmax* may not be sound. In practice, one-versus-rest is still often used.

If the training dataset is relatively balanced overall, one-versus-one will not suffer from classes imbalances. However, it is possible for classes to receive the same number of votes.

There exist other approaches, such as those based on error-correcting codes, although these too decompose the problem into many binary classifications, so cannot learn correlations between each of the classes.

Many algorithms are naturally multi-class, such as multi-layer perceptrons with a *softmax* output. Others can be generalized to a k-class setting using techniques specific to an underlying algorithm, like the SAMME algorithm for multi-class AdaBoost, which we will investigate later.

2 Polynomial kernel perceptron (one-versus-rest)

We have seen the general approach for a one-versus-rest kernel perceptron, but we now describe it in more detail, including the update rule.

For the handwritten digit recognition problem, we will train 10 binary kernel perceptron classifiers, each treating a unique digit as the positive class and all others as negative. We will store one $\boldsymbol{\alpha}^{(i)}$ vector per classifier i in a matrix $\boldsymbol{\alpha}$ of size $10 \times num\_examples$, in addition to all the training examples and their labels.

For each example we see, we will predict the positive class of the classifier with the greatest confidence, or

$$\hat{y}_t = \arg\max_k \sum_{i=1}^{m} \boldsymbol{\alpha}_i^{(k)} y_i K(\boldsymbol{x}_i, \boldsymbol{x}),$$

where $\boldsymbol{\alpha}^{(k)}$ is the row of the matrix corresponding to the k^{th} classifier. If the prediction is correct, no updates are performed. If the prediction is incorrect, we increment the index of $\boldsymbol{\alpha}^{(y_t)}$ corresponding to the example $\boldsymbol{x_t}$ (raising the confidence of correct classifier), and decrement its index in $\boldsymbol{\alpha}^{(\hat{y}_t)}$ (lowering the confidence of the incorrect classifier).

Algorithm 3 formalises this. As usual, the *for* loop can be repeated until a convergence criterion is met.

Algorithm 3: One-versus-rest kernel perceptron

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$

1. Initialize $\boldsymbol{\alpha} = 0_{k,m}$
2. **for** *t=1 to m* **do**
   - Receive pattern: $\boldsymbol{x}_t \in R^n$.
   - Predict: $\hat{y}_t = \arg\max_k \sum_{i=1}^{m} \boldsymbol{\alpha}_i^{(k)} y_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)$
   - Receive label: y_t
   - **if** $\hat{y}_t y_t \leq 0$ **then**
     - $\boldsymbol{\alpha}_t^{(y_t)} = \boldsymbol{\alpha}_t + 1$
     - $\boldsymbol{\alpha}_t^{(\hat{y}_t)} = \boldsymbol{\alpha}_t - 1$

end

As we are in a batch setting, we also pre-compute the Gram matrix in the initialisation step, which contains the results of applying the kernel function between every pair of examples in the training dataset. This saves on unnecessarily re-computing kernel functions during each iteration of the for loop, at the cost of $O(m^2)$ space.

We are now ready to implement a multi-class kernel perceptron with our handwritten digit dataset. We first choose to use a polynomial kernel function, given by

$$K_d(\boldsymbol{p}, \boldsymbol{q}) = (\boldsymbol{p} \cdot \boldsymbol{q})^d,$$

where d controls the degree of the polynomial. For $d = 1$, the kernel computes the dot product between example features. For $d > 1$, it computes the dot product for combinations of features.

(i) Basic results

We trained our one-versus-rest polynomial kernel perceptron until convergence with degrees $d = 1, ..., 7$ over 20 runs, using random 80% training data and 20% test data splits each run. We define an *epoch* as one round of the *for* loop. We use an early stopping convergence criterion, where the algorithm would stop if it had trained for a minimum of 10 epochs, and the mean training accuracy (measured by number of mistakes) of the last 5 epochs was less than 0.01 greater than the 5 epochs prior to those. We report the mean train and test errors over the runs for each d in Table 1.

The polynomial kernel with degree 1 (i.e. linear kernel) had the largest mean test error, 8.31. The kernels of other degrees are more difficult to distinguish, with mean test errors between 3 and 4.

Polynomial degree	**Train error**	**Test error**
1	5.84 ± 0.202	8.31 ± 1.39
2	0.395 ± 0.146	3.94 ± 0.541
3	0.160 ± 0.0976	3.24 ± 0.382
4	0.120 ± 0.0737	3.23 ± 0.412
5	0.084 ± 0.0444	3.18 ± 0.223
6	0.0558 ± 0.0491	3.15 ± 0.460
7	0.0578 ± 0.0376	3.31 ± 0.347

Table 1: The mean train and test error percentages and their standard deviations for a one-versus-rest polynomial kernel perceptron with degrees $d = 1, ..., 7$ over 20 runs

As previously discussed, we may be interested in how sparse the $\boldsymbol{\alpha}$ vectors are, as a greater number of non-zero elements will lead to a greater computation time. Table 2 shows the number of non-zero elements per classifier for degrees $d = 1, ..., 7$, over a single run. The rough trend is that a higher degree led to more sparse vectors, but the hypotheses are also more complex.

Polynomial degree	**Classifier**									
	0	1	2	3	4	5	6	7	8	9
1	402	248	675	657	737	763	460	519	761	681
2	179	166	307	306	418	311	197	268	341	350
3	151	117	227	224	297	239	160	197	239	256
4	125	107	199	202	256	213	140	178	222	224
5	127	89	174	191	196	197	132	135	195	169
6	125	97	177	182	232	178	123	148	190	184
7	104	101	169	179	231	173	120	135	197	182

Table 2: The number of non-zero $\boldsymbol{\alpha}$ vector elements for a one-versus-rest polynomial kernel perceptron classifier with degrees $d = 1, ..., 7$ over a single run

(ii) Cross-validation

In order to find the polynomial degree with the best performance on test data, we performed 20 runs of 5-fold cross-validation with degrees $d = 1, ..., 7$, on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 3.

Table 4 shows that these results have a mean test error of 3.23 and a mean polynomial degree of 5.05. The modal polynomial degree was 5, with 8 occurrences.

Polynomial degree	Test error
5	3.39
6	3.17
5	2.31
7	3.71
4	4.14
6	2.74
5	3.39
4	3.66
6	4.03
5	3.06
5	3.23
5	2.90
6	2.78
4	3.28
6	3.66
3	3.23
5	2.90
4	2.85
5	3.49
5	2.69

Table 3: The best polynomial degree after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	Mean polynomial degree
3.23 ± 0.468	5.05 ± 0.945

Table 4: The mean of the best polynomial degrees and test errors

(iii) Confusion matrix

We may be interested in understanding the kinds of errors of classifiers are making. For example, it seems likely that the digit 0 may sometimes be mistaken for a 6, because they are written similarly, with a hole in the middle and a similar round shape. We can investigate these kinds of issues with a confusion matrix, which shows the proportion of times each class was mistaken for each other class.

As in the scenario above, we performed 20 runs of 5-fold cross-validation with degrees $d = 1, ..., 7$, on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset.

Each test step produced a 10×10 matrix of counts, of the number of times each true class was mistakenly predicted to be another class. We then normalize each row, dividing each column in the class by the total number of times the true class was misclassified, in order to obtain proportions, indicating which classes were most likely to be mistaken as another. We take the mean and standard deviation of the matrices across the 20 runs to obtain the matrix shown in Table 5.

We find that the confusions seem to be as expected. For example, when a digit 3 was misclassified, it was most frequently as a digit 5, which is written very similarly. The digits 4 and 9 are also often confused with each other. There are many other examples recorded in the table with relatively high proportions, but none seem particularly surprising.

There do not seem to be any obvious confusion-related problems resulting from the class imbalances of digits 0 and 1.

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	0	0.00714 ± 0.0311	0.126 ± 0.171	0.144 ± 0.18	0.104 ± 0.155	0.141 ± 0.157	0.277 ± 0.27	0.0125 ± 0.0545	0.0789 ± 0.142	0.0592 ± 0.112
1	0.025 ± 0.109	0	0.104 ± 0.246	0 ± 0	0.212 ± 0.273	0.0125 ± 0.0545	0.221 ± 0.279	0.1 ± 0.255	0.075 ± 0.179	0 ± 0
2	0.115 ± 0.114	0.0539 ± 0.0897	0	0.15 ± 0.133	0.164 ± 0.103	0.0564 ± 0.081	0.0238 ± 0.062	0.278 ± 0.169	0.143 ± 0.0967	0.0156 ± 0.0487
3	0.0644 ± 0.101	0.00455 ± 0.0198	0.178 ± 0.119	0	0.0145 ± 0.0469	0.403 ± 0.141	0 ± 0	0.126 ± 0.113	0.198 ± 0.141	0.0113 ± 0.0352
4	0.0189 ± 0.0454	0.248 ± 0.219	0.161 ± 0.154	0.00937 ± 0.0298	0	0.0507 ± 0.0912	0.142 ± 0.12	0.103 ± 0.11	0.0149 ± 0.037	0.252 ± 0.202
5	0.205 ± 0.149	0.00625 ± 0.0272	0.0527 ± 0.0656	0.291 ± 0.152	0.0894 ± 0.108	0	0.162 ± 0.141	0.0408 ± 0.0695	0.0608 ± 0.0604	0.0922 ± 0.0886
6	0.362 ± 0.264	0.0829 ± 0.132	0.0755 ± 0.128	0 ± 0	0.162 ± 0.234	0.226 ± 0.181	0	0 ± 0	0.0792 ± 0.151	0.0125 ± 0.0545
7	0 ± 0	0.0621 ± 0.102	0.0938 ± 0.138	0.0905 ± 0.151	0.296 ± 0.203	0.028 ± 0.0689	0 ± 0	0	0.125 ± 0.174	0.304 ± 0.194
8	0.101 ± 0.0873	0.0733 ± 0.0812	0.143 ± 0.121	0.25 ± 0.147	0.0833 ± 0.0752	0.196 ± 0.111	0.0481 ± 0.0637	0.0841 ± 0.0991	0	0.0201 ± 0.0417
9	0.0522 ± 0.0925	0.0213 ± 0.0537	0.0183 ± 0.0553	0.0633 ± 0.119	0.362 ± 0.3	0.0335 ± 0.0729	0.0125 ± 0.0545	0.42 ± 0.219	0.0171 ± 0.0522	0

Table 5: Normalized confusion matrix

(iv) Hardest to predict data items

We used the twenty classifiers selected via cross-validation that were trained on an 80% training data split to find the hardest to predict data items. We tested each of the classifiers on the full 100% dataset and counted the number of times each example was misclassified.

Over the twenty runs of classifiers selected via cross-validation, 568 distinct examples were misclassified of a total data set size of 9,298. The five most misclassified examples were all misclassified at least eight times, and are shown in Figure 6, while the remaining examples were misclassified seven or fewer times.

It is not surprising that the algorithm found these difficult to predict. One example, which was perhaps mislabelled a 4 instead of a 1, was misclassified every run — it seems plausible that the classifier for digit *1-versus-rest* would have greater confidence that it was a 1 than the *4-versus-rest* classifier.

The other four examples are also difficult for a human to recognize as the given dataset labels.

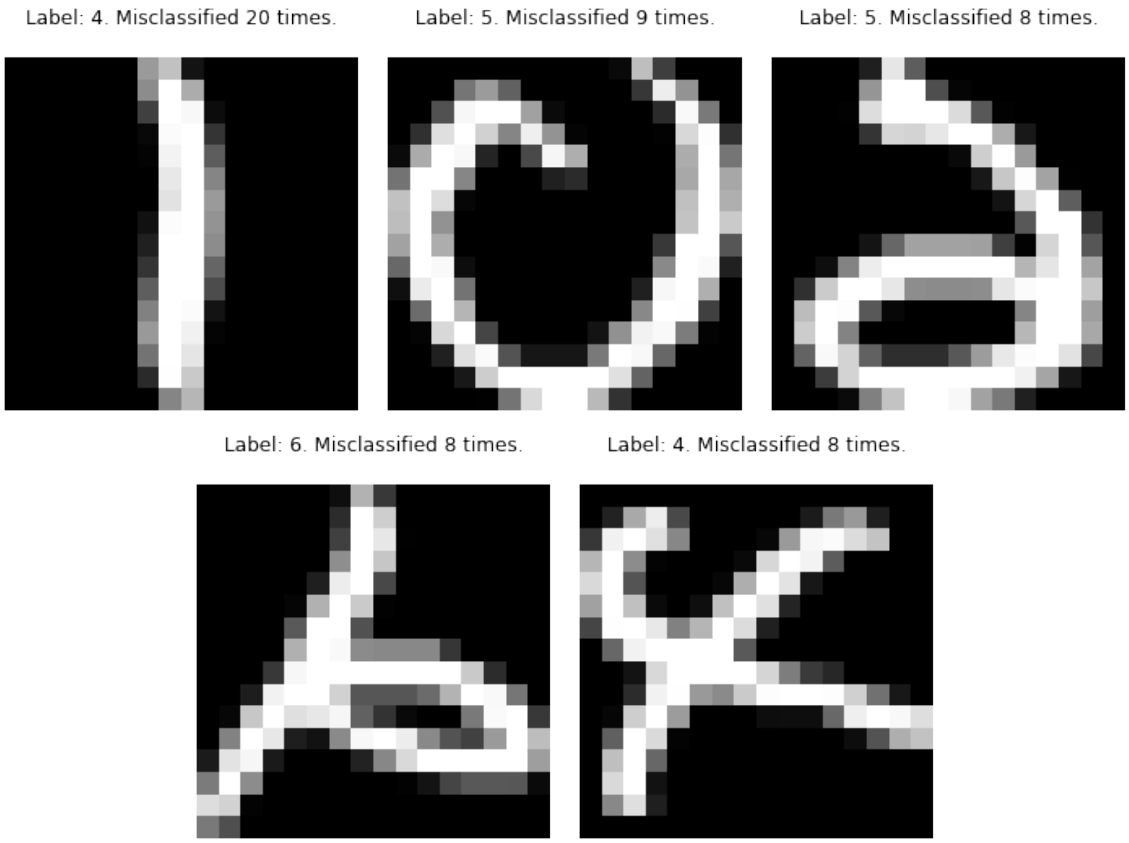


Figure 6: The five hardest to predict data items, based on 20 runs of selecting the best classifier via 5-fold cross-validation trained on an 80% training data split, testing on the entire dataset, and counting the number of mistakes per example

3 Gaussian kernel perceptron (one-versus-rest)

We will now use the same approach with a Gaussian kernel, given by

$$K(\boldsymbol{p}, \boldsymbol{q}) = e^{-c||\boldsymbol{p}-\boldsymbol{q}||^2},$$

where c is a width parameter that controls how much influence a training example has on the example it is predicting.

All else equal, if the width is small, the kernel function will be larger, so two examples will be considered more similar, and training examples that are far away have more influence. If the width is large, the kernel

function will be small, and the training example may be deemed to have less of a similarity to nearby examples. If the width is too small or too large, this may lead to underfitting or overfitting, respectively.

(i) Basic results

We trained our one-versus-rest Gaussian kernel perceptron until convergence with widths

$$c = 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-2}, 10^{-1}, 10^{0},$$

after some initial exploration to select these values. We trained over 20 runs, using random 80% training data and 20% test data splits each run. We use an early stopping convergence criterion, where the algorithm would stop if it had trained for a minimum of 10 epochs, and the mean training accuracy (measured by number of mistakes) of the last 5 epochs was less than 0.01 greater than the 5 epochs prior to those. We report the mean train and test errors over the runs for each width in Table 1.

The Gaussian kernel with width 0.01 had the smallest mean test error of 3.15, and the range of widths between 0.001 and 0.1 performed fairly well. Widths 0.00001 and 0.0001 had large mean test errors of 18.7 and 10.8, struggling to learn from the training data. Examples that were far away may have been deemed too similar, causing the classifiers to underfit.

Kernel width	**Train error**	**Test error**
0.00001	18.2 ± 0.589	18.7 ± 6.39
0.0001	10.7 ± 0.462	10.8 ± 3.92
0.001	3.26 ± 0.409	5.69 ± 1.10
0.01	0.11 ± 0.067	3.15 ± 0.367
0.1	0 ± 0	4.97 ± 0.358
1	0 ± 0	6.10 ± 0.597

Table 6: The mean train and test error percentages and their standard deviations for a one-versus-rest Gaussian kernel perceptron with various widths over 20 runs

Table 11 shows the number of non-zero elements per classifier for the different widths over a single run. This seems to clarify that indeed, examples that are far away from each other are now having too much influence. The vectors seem more sparse for the widths that generalized better to the test data.

Kernel width	**Classifier**									
	0	1	2	3	4	5	6	7	8	9
0.00001	3457	2027	3582	3361	3571	3636	3215	2910	3707	3444
0.0001	1358	715	1637	1451	1648	1723	1283	1202	1712	1712
0.001.	517	323	840	727	945	884	548	621	863	879
0.01	134	108	216	214	265	239	150	150	254	223
0.1	149	74	175	213	216	220	119	147	171	194
1	169	77	200	219	236	245	163	198	200	241

Table 7: The number of non-zero $\boldsymbol{\alpha}$ vector elements for a one-versus-rest Gaussian kernel perceptron with various widths over a single run

(ii) Cross-validation

The previous section showed that kernel widths between 0.001 to 0.1 led to the smallest test errors, with 0.01 being particularly good, so we chose to cross-validate within that range using widths $c = 2^{-9}, ..., 2^{-4}$ (minimum $2^{-9} = 0.00195$ and maximum $2^{-4} = 0.0625$).

In order to find the kernel width with the best performance on test data, we performed 20 runs of 5-fold cross-validation on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 8.

Table 9 shows that these results have a mean test error of 4.14 and a mean polynomial degree of 0.0145. The modal kernel width was $2^{-6} = 0.015625$, with 17 occurrences. This test error is greater than the best performing width in our basic results, so it seems like the Gaussian kernel perceptron is very sensitive to the width, and we could cross-validate over values even closer to 0.01 for better results.

Kernel width	**Test error**
0.007812	4.09
0.015625	4.46
0.015625	5.59
0.015625	3.71
0.015625	3.33
0.015625	3.87
0.015625	4.84
0.015625	4.41
0.015625	3.49
0.015625	3.50
0.007812	3.98
0.015625	3.98
0.015625	3.39
0.015625	4.46
0.007812	4.09
0.015625	4.68
0.015625	4.62
0.015625	4.46
0.015625	4.25
0.015625	3.66

Table 8: The best Gaussian kernel width after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	**Mean kernel width**
4.14 ± 0.567	0.0145 ± 0.00286

Table 9: The mean of the best Gaussian kernel widths and test errors

4 Polynomial kernel perceptron (one-versus-one)

Next, we will implement a one-versus-one polynomial kernel perceptron, which we discussed earlier. Here, we will describe it in more detail, including the update rule.

For a k-class problem, we will need to train $k(k-1)/2$ binary classifiers. For the 10-class handwritten digit recognition problem, this means training 45 classifiers, where each of the 10 digits is the positive or negative class in 9 of the 45 classifiers. The set up is similar to before, with each row $\boldsymbol{\alpha}^{(i)}$ in a matrix $\boldsymbol{\alpha}$ of size $45 \times \textit{num\_examples}$ corresponding to a single classifier. We again store all the training examples and their labels, and pre-compute the Gram matrix.

For each example we see, each of the 45 classifiers will make a positive or negative prediction, corresponding to a class. The overall prediction is then the class with the greatest number of individual predictions, or votes. If any of the 10 classifiers that have the true label as the positive or negative class are incorrect, an update is performed on each of them.

If the true label y is the positive class for the incorrect classifier, we increment the index of $\boldsymbol{\alpha}^{(y)}$ corresponding to the example; if it is the negative class, we decrement this. This raises the confidence of the correct predictions in future.

Algorithm 4 formalises this. As usual, the *for* loop can be repeated until a convergence criterion is met.

Algorithm 4: One-versus-one kernel perceptron

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$

1. Initialize $\boldsymbol{\alpha} = 0_{k(k-1)/2,m}$
2. **for** *t=1 to m* **do**
   - Receive pattern: $\boldsymbol{x}_t \in R^n$.
   - Evaluate: $sign \sum_{i=1}^{m} \boldsymbol{\alpha}_i^{(k)} y_i K(\boldsymbol{x}_i, \boldsymbol{x}_t)$, for each binary classifier, and convert each output to one of the k classes to get $\hat{y}_t^{(i)}$
   - Predict: $\hat{y}_t$ = the class with the most votes
   - Receive label: y_t
   - Update: **for** $i = 1$ *to* $k(k-1)/2$ **do**
     - **if** *classifier involves label* y_t *AND* $\hat{y}_t^{(i)} sign \cdot y_t \leq 0$ **then**
       - $\boldsymbol{\alpha}_t^{(i)} = \boldsymbol{\alpha}_t^{(i)} + 1$ if the true label is the positive class for classifier
       - $\boldsymbol{\alpha}_t^{(i)} = \boldsymbol{\alpha}_t^{(i)} - 1$ if the true label is the negative class for classifier
   - **end**

end

(i) Basic results

We trained our one-versus-one polynomial kernel perceptron until convergence with degrees $d = 1, ..., 7$ over 20 runs, using random 80% training data and 20% test data splits each run. After some initial exploration, we discovered that fewer training epochs were needed to reach convergence for one-versus-one. Therefore, we use an early stopping convergence criterion, where the algorithm would stop if it had trained for a minimum of 2 epochs, and the mean training accuracy (measured by number of mistakes) of the last 2 epochs was less than 0.01 greater than the 2 epochs prior to those. We report the mean train and test errors over the runs for each d in Table 1.

Again, the polynomial kernel with degree 1 (i.e. linear kernel) had the largest mean test error, 6.40. The kernels of other degrees are more difficult to distinguish, with mean test errors between 3 and 4. Notably, the best performing degree was 4, which is a lower degree than for the one-versus-rest classifier. As we will see below, the number of non-zero $\boldsymbol{\alpha}$ coefficients in the one-versus-one classifiers is much lower than for one-versus-rest, so they are much smaller and simpler, perhaps illustrated by the best performing degree being smaller, with less complexity is required to represent the one-versus-one classifiers.

Polynomial degree	Train error	Test error
1	3.61 ± 0.373	6.40 ± 0.728
2	0.547 ± 0.199	3.66 ± 0.469
3	0.241 ± 0.0701	3.48 ± 0.404
4	0.111 ± 0.0358	3.15 ± 0.42
5	0.109 ± 0.0416	3.42 ± 0.444
6	0.0827 ± 0.0304	3.39 ± 0.488
7	0.082 ± 0.0619	3.66 ± 0.625

Table 10: The mean train and test error percentages and their standard deviations for a one-versus-one polynomial kernel perceptron with degrees $d = 1, ..., 7$ over 20 runs

Table 11 shows the number of non-zero $\boldsymbol{\alpha}$ vector elements for the one-versus-one polynomial kernel perceptron classifiers over a single run, for a sample of the first 15 of 45 classifiers.

The values for the classifiers are similar, but not shown for succinctness. Compared to the numbers in for the one-versus-rest polynomial kernel perceptron in Table 2, this numbers are greatly reduced, confirming our intuition that the classifiers would be smaller in a one-versus-one setting, i.e. they would rely on fewer examples, and therefore be quicker to make predictions at test time.

Degree	Classifier														
	0-1	0-2	0-3	0-4	0-5	0-6	0-7	0-8	0-9	1-2	1-3	1-4	1-5	1-6	1-7
1	34	121	82	72	164	116	49	89	52	52	44	89	54	57	46
2	21	68	50	51	73	72	28	50	29	34	19	54	28	32	26
3	14	64	45	31	69	57	20	38	21	22	21	57	27	23	29
4	16	64	44	38	60	55	17	41	20	21	16	48	22	21	22
5	15	65	38	29	63	57	23	44	25	27	21	54	21	29	28
6	16	62	49	36	63	64	27	49	27	22	17	55	21	27	20
7	16	54	46	30	64	56	24	50	23	25	17	47	22	28	13

Table 11: The number of non-zero $\boldsymbol{\alpha}$ vector elements for a one-versus-one polynomial kernel perceptron with degrees $d = 1, ..., 7$ over a single run, for the first 15 of 45 classifiers

(ii) Cross-validation

In order to find the polynomial degree with the best performance on test data, we performed 20 runs of 5-fold cross-validation with degrees $d = 1, ..., 7$, on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 12.

Table 13 shows that these results have a mean test error of 3.70 and a mean polynomial degree of 3.90. The modal polynomial degree was 5, with 8 occurrences.

The mean polynomial degree is significantly smaller (3.90) than for the cross-validation performed on the same range for one-versus-rest (5.05) as shown in Table 4. As discussed in the previous section, the one-versus-one classifiers seem to be smaller and suggest that less complex hypotheses are required to represent the classification problems, hence the lower mean polynomial degree. The mean test error is marginally higher here, at 3.23 versus 3.90.

Polynomial degree	Test error
4	3.12
3	3.50
4	3.28
4	3.66
3	3.23
4	3.06
4	3.76
3	3.55
4	3.71
4	3.87
2	3.55
5	3.49
4	2.80
4	3.39
3	3.49
5	3.66
5	3.39
5	3.71
4	4.09
4	7.10

Table 12: The best polynomial degree after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	Mean polynomial degree
3.70 ± 0.860	3.90 ± 0.788

Table 13: The mean of the best polynomials degrees and test errors

5 Gaussian kernel SVM (one-versus-rest)

In this section, we will implement **two** versions of a multi-class support vector machine (SVM) with a Gaussian kernel, using the one-versus-rest strategy. There also exist methods to solve a multi-class formulation of the SVM directly, rather than decompose it into multiple binary classification problems [3].

The **first** version will solve the quadratic programming problem with the Python library *CVXOPT*, but this approach is too slow for our purposes to train on the entire dataset. Therefore, we will implement a **second** version using sequential minimal optimization (SMO), which solves much more quickly.

In its basic form, the *hard-margin* linear SVM, the model performs binary classification on linearly separable data. The algorithm learns a hyperplane with the maximum possible margin that is halfway between the two classes, determined by the examples in each class that are closest, called the support vectors, as depicted in Figure 7.

Given training examples $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$, the equation for a hyperplane is given by

$$\boldsymbol{w}^T \boldsymbol{x} - b = 0.$$

If the data are linearly separable, the choice of $\boldsymbol{w}$ and b which maximizes the margin is not yet unique, as scaling by a positive constant gives the same hyperplane. Therefore, we add an additional constraint and work with the *canonical* hyperplane, which requires that the margin from the hyperplane to the support vectors is 1, i.e.

$$\min_{i=1}^{m} y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1,$$

for which $1/||w||$ is the margin.

As we wish to maximize the margin, the hard-margin problem can then be formulated as

$$\begin{aligned} \text{Minimize} \quad & \frac{1}{2}\boldsymbol{w}^T \boldsymbol{w}, \ (\boldsymbol{w} \in R^n) \\ \text{s.t.} \quad & y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1, \ i = 1, ..., m. \end{aligned} \tag{1}$$

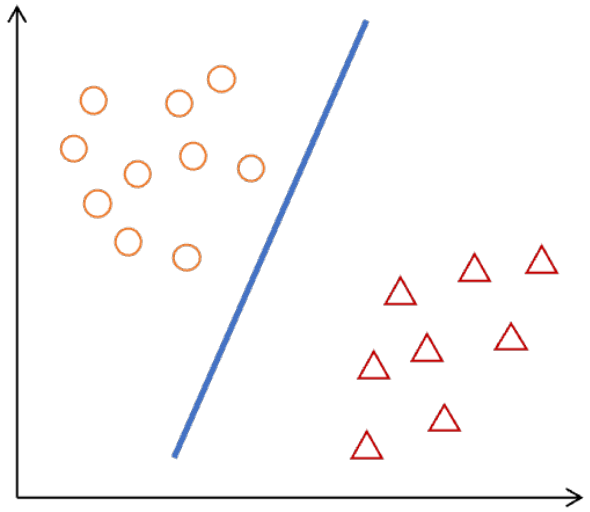
For many problems, data are not linearly separable, and we can formulate a *soft-margin* SVM, which allows some slack, i.e. for some training examples to be misclassified. We introduce a parameter $C > 0$ that trades-off the amount of slack with the margin, penalizing training examples which are on the wrong side of it. A convex, *hinge* loss function tends to be used for the penalty, denoted ξ in Figure 8, and is given by

$$max(0, 1 - y_i(\boldsymbol{w}^T \boldsymbol{x}_i - b)),$$

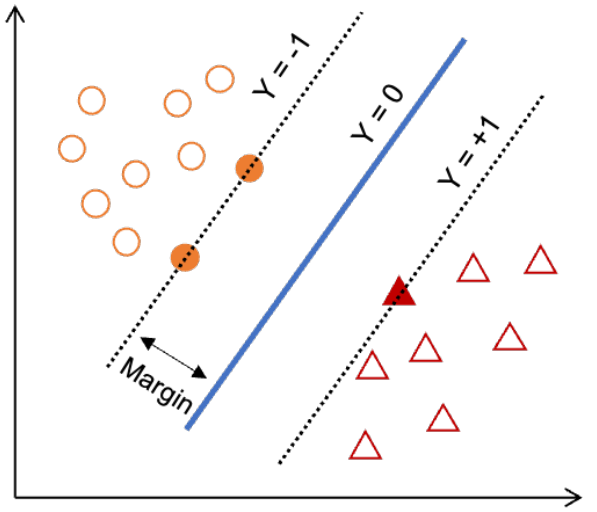
which equals 0 for examples on the correct side of the margin, and increases linearly with the distance from the margin for examples on the incorrect side. As $C \to \infty$, fewer mistakes are allowed and the problem tends towards the hard-margin setting. If C is small, more mistakes are allowed, so the training error increases, but the model may generalize better when put to use. If C is infinity, it is equivalent to the hard-margin setting. C is often selected using leave-one-out cross-validation.

The optimization problem can then be formulated as

$$\begin{aligned} \text{Minimize} \quad & \frac{1}{2}\boldsymbol{w}^T \boldsymbol{w} + C \sum_{n=1}^{N} \xi_n, \ (\boldsymbol{w} \in R^n) \\ \text{s.t.} \quad & y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) \geq 1 - \xi, \\ & \xi_i \geq 0, \quad i = 1, ..., m. \end{aligned} \tag{2}$$

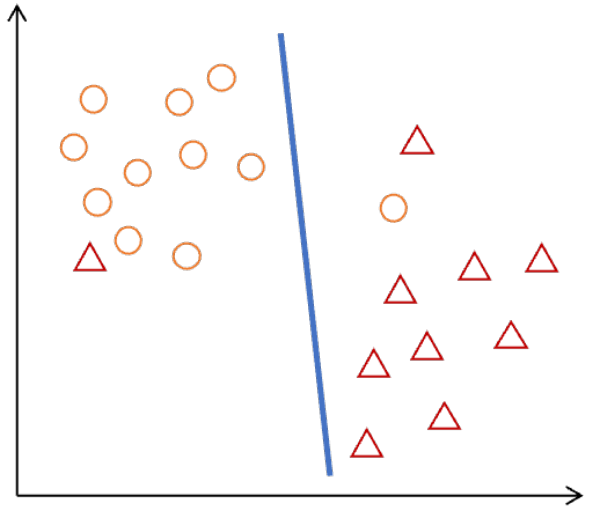


(a) A linearly separable dataset — the goal of the hard-margin SVM is to find a hyperplane (in blue) which maximizes the margin between the examples in the two class

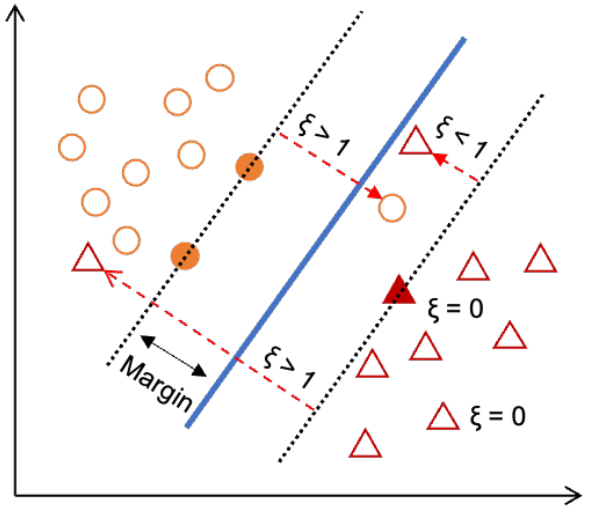


(b) The hard-margin SVM finds the with maximum-margin hyperplane, halfway between the two classes — the support vectors, which determine the hyperplane, are filled

Figure 7



(a) A non-linearly separable dataset — it is not possible to fit a hyperplane that separates the two classes



(b) The soft-margin SVM finds a maximum-margin hyperplane while allowing some training examples to be misclassified, according to C

Figure 8

This is a convex optimization problem, and can be solved via the Lagrangian, given by

$$L(\boldsymbol{w}, \boldsymbol{\xi}, b, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(\boldsymbol{w}^T\boldsymbol{x}_i + b) + \xi_i - 1] - \sum_{i=1}^{m}\beta_i\xi_i, \tag{3}$$

where $\alpha_i, \beta_i \geq 0$ are the Lagrange multipliers, and which we minimize over $\boldsymbol{w}$, $\boldsymbol{\xi}$, and b, and maximize over $\boldsymbol{\alpha}, \boldsymbol{\beta} \geq 0$. If we differentiate with respect to $\boldsymbol{w}$, $\boldsymbol{\xi}$, and b as follows,

$$\begin{aligned}
\frac{\partial L}{\partial b} &= -\sum_{i=1}^{m} y_i \alpha_i = 0 \\
\frac{\partial L}{\partial \boldsymbol{w}} &= \boldsymbol{w} - \sum_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i = \mathbf{0} \implies \boldsymbol{w} = \sum_{i=1}^{m} \alpha_i y_i x_i \\
\frac{\partial L}{\partial \xi_i} &= C - \alpha_i - \beta_i = 0 \implies 0 \leq \alpha_i \leq C,
\end{aligned} \tag{4}$$

we can substitute these back in to Equation 3 to obtain a dual form. This is a quadratic programming problem given by

$$\begin{aligned}
\text{Maximize} \quad & Q(\alpha) := -\frac{1}{2}\boldsymbol{\alpha}^T A \boldsymbol{\alpha} + \sum_i \alpha_i \\
\text{s.t.} \quad & \sum_i y_i \alpha_i = 0, \\
& 0 \leq \alpha_i \leq C, \quad i = 1, ..., m,
\end{aligned} \tag{5}$$

where $\bar{\boldsymbol{w}} = \sum_{i=1}^{m} \bar{\alpha}_i y_i \boldsymbol{x}_i$ is implicitly represented as a linear combination of the training examples $\boldsymbol{x}_i$, and α_i are the Lagrange multipliers which dictate that their corresponding training example $\boldsymbol{x}_i$ is a support vector if positive. Support vectors for which $0 < \alpha_i < C$ lie on the margin, whereas we call those that lie within the margin *bounded* support vectors, where $\alpha_i = C$. The bias term, $\bar{b}$, can then be calculated using the support vectors as

$$b = \frac{1}{N_M} \sum_{i \in M} y_i - \sum_{j \in S} \alpha_j y_j \boldsymbol{x}_i \boldsymbol{x}_j,$$

where S is the indices of the support vectors, and M the indices of those with $0 < \alpha_n < C$.

As with the dual form of the perceptron, we can then obtain a non-linear, soft-margin SVN by applying a feature map to the examples which is consistent with a kernel function $K(\boldsymbol{x}_t, \boldsymbol{x}) = \phi(\boldsymbol{x}_t)^T \phi(\boldsymbol{x})$, so that the implicit $\boldsymbol{w}$ equals $\sum_{i=1}^{m} \bar{\alpha}_i y_i \phi(\boldsymbol{x}_i)$ in the optimization problem given by Equation 2.

(i) Approach 1: Gaussian Kernel SVM with CVXOPT

In our first version of this classifier, we implement a soft-margin, Gaussian kernel SVM which solves this quadratic programming problem using the Python library *CVXOPT*. We find that this version is inefficient compared to, and performs marginally worse than, our second version in the next section which uses sequential minimal optimization, but we record our results nonetheless, in case they are of interest.

We use a one-versus-rest approach to multi-class classification, training 10 binary SVMs. As discussed earlier, a limitation of this technique is that the classes (*one* and *rest*) in each binary SVM will be imbalanced, as the *rest* class contains training data for 9 different classes, so the resulting classifier confidences may not be calibrated. Techniques such as Platt scaling can be used to help solve this.

The time to train this SVM lies between quadratic, $O(n^2)$, and cubic, $O(n^3)$, in the number of training examples [1]. The time complexity depends on the number of support vectors, which is in turn determined by C; for a smaller C, fewer support vectors will be found, and the complexity grows linearly between $O(n^2)$ and $O(n^3)$ in the number of support vectors.

A number of other approaches have been proposed to solve the imbalance issue [2]. Since this version of the classifier is relatively slow to train in comparison to our next approach, and the test performance is still reasonably competitive (see Table 16), we choose a sub-sampling approach where to train each binary SVM we sample an equal number of *rest* examples (without replacement) as the number of *one* examples we have in our training dataset. This may help to balance the classifiers, and greatly speeds up the computation time. We note that in Approach 2 to this algorithm, each classifier is trained on all the data, but these results are recorded for interest.

If the training dataset as a whole is balanced, another solution is to use one-versus-one multi-class classification, which trains 45 binary classifiers for our problem, each on a much smaller number of examples.

As with the kernel perceptron, we pre-compute the Gram matrix, and we select the training examples with Lagrange multipliers $\alpha_i > 10^{-5}$ to be support vectors.

(ii) Basic results

We trained our one-versus-rest Gaussian kernel SVM until convergence with C values $C = 0.5, 1, 2, 4, 8$ and kernel widths width $= 2^{-5}, 2^{-6}, 2^{-7}$, after some initial exploration where we performed a single run over the hyperparameter combinations $C = 2^{-7}, ..., 2^{7}$ and width $= 2^{-7}, ..., 2^{7}$. We trained over 20 runs, using random 80% training data and 20% test data splits each run. We report the mean train and test errors over the runs for each combination of hyperparameters in Table 14.

The combinations with kernel widths $2^{-6} = 0.0156$ and C values of 4 and 8 had the smallest mean test errors of 2.94, with similar standard deviations. This is the same kernel width that was most frequently selected during Gaussian kernel perceptron cross-validation.

C	Kernel width	Train error	Test error
0.5	0.00781	2.67 ± 0.0813	4.26 ± 0.368
0.5	0.0156	1.66 ± 0.0733	3.70 ± 0.415
0.5	0.0312	1.05 ± 0.0917	7.37 ± 0.422
1	0.00781	1.68 ± 0.0921	3.51 ± 0.378
1	0.0156	0.558 ± 0.0502	3.23 ± 0.395
1	0.0312	0.143 ± 0.0201	5.00 ± 0.529
2	0.00781	0.709 ± 0.0721	3.27 ± 0.469
2	0.0156	0.174 ± 0.0219	2.98 ± 0.453
2	0.0312	0.045 ± 0.0115	4.67 ± 0.373
4	0.00781	0.426 ± 0.0729	3.20 ± 0.316
4	0.0156	0.145 ± 0.0392	2.94 ± 0.367
4	0.0312	0.0329 ± 0.00995	4.92 ± 0.363
8	0.00781	0.376 ± 0.079	3.08 ± 0.329
8	0.0156	0.149 ± 0.0396	2.94 ± 0.325
8	0.0312	0.0289 ± 0.0122	4.98 ± 0.384

Table 14: The mean train and test error percentages and their standard deviations for a one-versus-rest Gaussian kernel SVM with C values $C = 0.5, 1, 2, 4, 8$ and kernel widths $width = 2^{-5}, 2^{-6}, 2^{-7}$ over 20 runs

Table 15 shows the support vectors per classifier for the different combinations of C and kernel width over a single run. Perhaps surprisingly, varying C did not have a noticeable effect on the number of support vectors, particularly relative to the kernel width, which had a large effect. We find that the best performing combinations (with a width of 0.0156) had an in-between number of support vectors; this may have allowed it to reach a sweet spot between underfitting and overfitting.

C	Kernel width	Classifier									
		0	1	2	3	4	5	6	7	8	9
0.5	0.00781	411	120	518	402	416	453	316	290	445	368
0.5	0.0156	754	153	750	620	667	646	531	475	607	572
0.5	0.0312	1525	850	1309	1028	1096	973	942	868	918	897
1	0.00781	369	106	439	369	353	370	290	267	426	336
1	0.0156	689	193	713	578	629	634	504	443	613	553
1	0.0312	1468	829	1262	1032	1049	982	943	813	904	886
2	0.00781	367	127	437	348	352	399	292	257	418	338
2	0.0156	699	222	728	607	656	640	521	440	598	537
2	0.0312	1427	880	1317	1000	1129	986	911	824	925	904
4	0.00781	365	111	431	346	340	441	287	221	400	309
4	0.0156	718	296	743	617	648	644	505	451	615	538
4	0.0312	1525	848	1307	1009	1117	985	908	802	888	894
8	0.00781	342	105	421	331	318	410	290	237	408	302
8	0.0156	711	247	755	599	640	637	527	450	603	535
8	0.0312	1454	866	1299	1017	1118	1004	909	815	903	891

Table 15: The number of support vectors for a one-versus-rest Gaussian kernel SVM with various values of C and widths over a single run

(iii) Cross-validation

We have seen that the kernel width of 2^{-6} performed best in the basic results, so we will perform cross-validation around that range with values $2^{-5}, 2^{-6}, 2^{-7}$. The C values 0.5 and 1 performed the worst in our basic results, so will ignore those and cross-validate over $C = 2, 4, 8$.

In order to find the combination with the best performance on test data, we performed 20 runs of 5-fold cross-validation on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 16.

Table 23 shows that these results have a mean test error of 2.89, a mean kernel width of 0.0152. The modal kernel width was $2^{-6} = 0.015625$, with 19 occurrences. This seemed to matter more than the value of C, as each was represented multiple times in the results, and which had a mean and standard deviation of 4.20 ± 2.67. Despite Approach 1 to the soft-margin kernel SVM training on subsamples of the data, the this test performance is the best so far. In Approach 2, we will train on all the data, leading to even better test performance.

C	Kernel width	Test error
2	0.0156	2.74
8	0.0156	2.85
2	0.0156	3.01
4	0.0156	2.90
2	0.0156	3.01
4	0.0156	3.23
8	0.0156	2.58
4	0.0156	2.58
4	0.0156	3.06
2	0.00781	2.96
8	0.0156	2.80
2	0.0156	2.63
2	0.0156	3.01
2	0.0156	3.06
8	0.0156	2.85
2	0.0156	2.58
8	0.0156	3.06
2	0.0156	3.55
2	0.0156	2.37
8	0.0156	2.96

Table 16: The best C and kernel width after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	Mean C	Mean kernel width
2.89 ± 0.267	4.20 ± 2.67	0.0152 ± 0.00174

Table 17: The mean of the best C values, kernel widths, and test errors

(iv) Approach 2: Gaussian Kernel SVM with SMO

In 1998, Platt introduced the sequential minimal optimization (SMO) algorithm for solving the SVM quadratic programming problem [7], which we will implement. In the standard SMO algorithm, pairs of Lagrange multipliers are repeatedly selected and optimized analytically (as opposed to optimizing all multipliers using numerical optimization in the quadratic programming approach), chosen in an order that accords to those which should maximize the optimization problem at each step, until the algorithm reaches global convergence. We find that this approach is much more computationally tractable for our purposes, and train each classifier on all of the data.

We implement a simplified SMO algorithm which instead iterates through the Lagrange multipliers, based on the Stanford CS229 course notes [8], rather than using a selection process. This adaptation does not guarantee global convergence, but works well in practice for our purposes. We can choose to say that the algorithm has converged if it passes through each combination of multipliers a number of times without making an update, and we can implement early stopping, where we train for a maximum number of passes if it has not yet converged.

(v) Basic results

We performed some initial exploration, with a single run over $C = 2^{-7}, ..., 2^7$ and width $= 2^{-7}, ..., 2^7$, and find that C values $C = 1, 2, 4$ and kernel widths $width = 2^{-5}, 2^{-6}, 2^{-7}$ are most worth exploring further, based on their test errors.

Similarly, we explored different combinations of hyperparameters for the convergence criteria: the number of consecutive passes without updates to say that the algorithm has converged, and the maximum number

of passes to train for if not. We tried values $2, 3$ for the former, and between 25 and 100 for the latter. Over a few runs, we found that performance did not improve enough relative to the training time for the latter, so chose a value of 40 maximum passes, and also 2 consecutive passes without updates.

We trained over 20 runs, using random 80% training data and 20% test data splits each run. We report the mean train and test errors over the runs for each combination of hyperparameters in Table 18.

The combination with kernel width $2^{-6} = 0.0156$ and $C = 4$ had the smallest mean test error of 2.08. This is the same kernel width that was most frequently selected during Gaussian kernel perceptron cross-validation.

C	Kernel width	Train error	Test error
1	0.00781	0.748 ± 0.0627	2.76 ± 0.397
1	0.0156	0.147 ± 0.024	2.27 ± 0.274
1	0.0312	0.045 ± 0.0115	3.33 ± 0.344
2	0.00781	0.227 ± 0.0429	2.37 ± 0.35
2	0.0156	0.0484 ± 0.0108	2.30 ± 0.29
2	0.0312	0.0208 ± 0.00793	2.79 ± 0.422
4	0.00781	0.0686 ± 0.0159	2.22 ± 0.263
4	0.0156	0.0255 ± 0.00723	2.08 ± 0.284
4	0.0312	0.0229 ± 0.00616	3.16 ± 0.417

Table 18: The mean train and test error percentages and their standard deviations for a one-versus-rest Gaussian kernel SVM with C values $C = 1, 2, 4$ and kernel widths $width = 2^{-5}, 2^{-6}, 2^{-7}$ over 20 runs

(vi) Cross-validation

In the basic results, we found that $C = 4$ was the best performer, and width $= 2^{-6}, 2^{-7}$ performed well. Again, the kernel width seems to be a more significant factor, and it seems worth exploring this range of values, so we chose to cross-validate over the combinations $C = 4$ and width $= 0.005, 0.01, 0.015$. We continue to use 40 maximum passes and 2 consecutive passes without updates for our convergence criteria, and note that it is possible that performance may be marginally improved by training for longer.

In order to find the combination with the best performance on test data, we performed 20 runs of 5-fold cross-validation on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 19.

Table 20 shows that these results have a mean test error of 2.12, a mean kernel width of 0.0145. The modal kernel width was 0.0015 with 18 occurrences. We find that this is our best performing algorithm.

C	Kernel width	Test error
4	0.015	2.26
4	0.015	2.15
4	0.015	2.58
4	0.015	2.26
4	0.015	2.42
4	0.015	2.26
4	0.015	2.20
4	0.015	2.04
4	0.015	1.77
4	0.01	2.47
4	0.015	1.83
4	0.01	2.10
4	0.015	2.26
4	0.015	1.83
4	0.015	2.20
4	0.015	1.77
4	0.015	1.88
4	0.015	2.42
4	0.015	1.99
4	0.015	1.77

Table 19: The best C and kernel width after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	Mean C	Mean kernel width
2.12 ± 0.254	4 ± 0	0.0145 ± 0.00154

Table 20: The mean of the best C values, kernel widths, and test errors

6 Multi-class AdaBoost (SAMME)

In Section 1, we described the existence of algorithms that could generalize to a multi-class setting using techniques specific to an underlying algorithm, as opposed to decomposing the problem into many binary classifications as in one-versus-rest and one-versus-one.

Here, we will describe and implement an algorithm called SAMME (Stagewise Additive Modeling using a Multi-class Exponential loss function) that extends the binary AdaBoost boosting algorithm to k-classes [4].

Boosting is a general term for algorithms which combine many weaker classifiers which have high biases to construct a single, stronger model, with less bias. In particular, we iteratively add *weak learners* to our model, which are algorithms that need only be marginally better than random guessing, and weight them according to their performance in the overall model, making predictions by weighted majority. Each training example also has an associated weight which is updated during each iteration, which focuses the training of subsequent weak learners on the examples which have been most difficult to classify.

The AdaBoost algorithm for binary classification works as follows. The user will specify the number and type of weak learners they wish to train. Initially, the training examples and labels are received, and their associated weights are all equal. There is one iteration per number of classifiers, which each fit a weak learner classifier of the chosen model, focusing on examples which currently have the most weight. The weak learner is also weighted according to its training error. This effectively adds a new basis function (a weak learner) with a weight to the overall model. At the end of the iteration, the training examples are re-weighted so that subsequent iterations focus on learning from the most difficult examples, and less on the easier ones. Prediction then works by majority vote. Algorithm 5 formalises this.

Algorithm 5: AdaBoost

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{-1, 1\}$

1. Initialize the sample weights $w_i = 1/n,\ i = 1, 2, ..., n$
2. **for** *m=1 to M* **do**
   (a) Fit a classifier $T^{(m)}(\boldsymbol{x})$ to the training data using weights w_i
   (b) Compute
   $$err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{I}(c_i \neq T^{(m)}(\boldsymbol{x}_i)) / \sum_{i=1}^{n} w_i.$$
   (c) Compute
   $$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}}. \tag{6}$$
   (d) Set
   $$w_i \leftarrow w_i \cdot \exp(\alpha^{(m)} \cdot \mathbb{I}(c_i \neq T^{(m)}(\boldsymbol{x}_i))),\ i = 1, ..., n.$$
   (e) Re-normalize w_i.

   end
3. Output
$$C(\boldsymbol{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{I}(T^{(m)}(\boldsymbol{x}) = k).$$

Boosting algorithms, like AdaBoost, often use decision trees are the underlying model. Decision trees recursively partition space according to features, creating a tree made up of nodes (for features of the examples), branches (for checking the values of those features), and leaves (for assigning a class to the example). In our case, the nodes would be specific pixels, and the branches would check whether their gray-scale value was above or below a certain threshold, which may have a bearing on its class — for example, we may expect the digit 0 to have an empty ellipse in the middle of the image.

There are various methods for decision trees to choose which features to create nodes and branches (called splits) on. The best splits tend to be the purest, i.e. decision rules that have the most (or all) of one class on one side, and the fewest (or none) of another class on the other, since this split provides the most information about the class of an example. We will use the Gini impurity, which takes examples at random from the training dataset, assigns them a new classification based on the base class distributions, and measures the frequency at which this example would be misclassified. The split which results in the lowest impurity, or highest purity, is selected. The formula is given by

$$G = \sum_{i=1}^{k} p(i) \cdot (1 - p(i)),$$

where k is the number of classes, and $p(i)$ is the probability of picking an example with class i.

In practice, decision trees with only a single split (depth 1), called decision stumps, are often used in boosting. In our experiments, we will investigate using trees with depths between 1 to 8.

We will now extend Algorithm 5 to a multi-class setting. AdaBoost added weak learners to the overall model if they were better than random guessing, which was equivalent to an accuracy rate greater than a half. In a multi-class setting, this target may be difficult to achieve, but we can instead aim for an accuracy rate greater than $1/k$, where k is the total number of classes. This only requires a simple change to Equation 6, and is called the SAMME algorithm, formalised in Algorithm 6.

Algorithm 6: SAMME

Input: $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_m, y_m) \in R^n \times \{1, ..., K\}$

1. Initialize the sample weights $w_i = 1/n,\ i = 1, 2, ..., n$
2. **for** *m=1 to M* **do**
   (a) Fit a classifier $T^{(m)}(\boldsymbol{x})$ to the training data using weights w_i
   (b) Compute
   $$err^{(m)} = \sum_{i=1}^{n} w_i \mathbb{I}(c_i \neq T^{(m)}(\boldsymbol{x}_i)) / \sum_{i=1}^{n} w_i.$$
   (c) Compute
   $$\alpha^{(m)} = \log \frac{1 - err^{(m)}}{err^{(m)}} + \log(K - 1).$$
   (d) Set
   $$w_i \leftarrow w_i \cdot \exp(\alpha^{(m)} \cdot \mathbb{I}(c_i \neq T^{(m)}(\boldsymbol{x}_i))),\ i = 1, ..., n.$$
   (e) Re-normalize w_i.

   end
3. Output
   $$C(\boldsymbol{x}) = \arg\max_k \sum_{m=1}^{M} \alpha^{(m)} \cdot \mathbb{I}(T^{(m)}(\boldsymbol{x}) = k).$$

The complexity of SAMME depends on the number of classifiers and the depth of the trees that are trained. The cost of training a decision tree is $O(dpnlog(n))$, where d is the maximum depth, p is the number of features per training example, n is the number of examples. As we are training M classifiers, the total cost is $O(dpnlog(n)M)$. This is more efficient than one-versus-rest and one-versus-one approaches, which are $O(dpnlog(n)MK)$ and $O(dpnlog(n)MK^2)$.

In a different form of the algorithm, called SAMME.R, the weak learners output confidence estimates as opposed to classifications. This is more fine-grained, and tends to converge more quickly than SAMME.

For our experiments, we code our own implementation of SAMME, but make use of the *DecisionTreeClassifier* class in *scikit-learn* as the underlying model, using the Gini impurity as described for splitting. We do not implement pruning, varying only the depth of the trees we create.

(i) Basic results

After some initial exploration, we decided to train our SAMME models with number of classifiers 25, 50, and 100, using decision trees as the underlying learners, with maximum depths 1, 2, 4, and 8. We performed 20 runs for each combination, using random 80% training data and 20% test data splits each run. We report the mean train and test errors over the runs for each d in Table 21.

Unsurprisingly, increasing the number of classifiers and the maximum depths led to a decrease in the mean test errors, and the best performing was the combination with the most of each, with a mean test error of 3.60. A greater number of classifiers can learn a more complex model, and a higher depth trains less weak learners to begin with.

Perhaps surprisingly, decision stumps (with a maximum depth of 1) performed fairly poorly, although the number of classifiers was also small. In a separate experiment, they roughly reached 85% test error with 500 classifiers or more. It would be interesting to see whether variants, like SAMME.R, performed better with stumps, as well as different splitting metrics.

Number of classifiers	Max depth	Train error	Test error
25	1	40.8 ± 5.84	41.3 ± 5.82
25	2	19.6 ± 1.73	20.9 ± 1.85
25	4	6.30 ± 0.536	10.5 ± 0.758
25	8	0 ± 0	5.20 ± 0.451
50	1	25.6 ± 2.56	26.0 ± 2.33
50	2	15.7 ± 1.33	17.0 ± 1.19
50	4	3.36 ± 0.389	8.20 ± 0.370
50	8	0 ± 0	4.10 ± 0.436
100	1	20.4 ± 1.58	21.2 ± 1.62
100	2	12.4 ± 0.974	14.3 ± 1.18
100	4	1.90 ± 0.254	6.50 ± 0.620
100	8	0 ± 0	3.60 ± 0.550

Table 21: The mean train and test error percentages and their standard deviations for the SAMME algorithm with number of classifiers $number\_of\_classifiers = 25, 50, 100$ and maximum depths $max\_depth = 1, 2, 4, 8$ over 20 runs

(ii) Cross-validation

The previous section showed that a greater number of classifiers and maximum tree depth led to the smallest test error, so it seems likely that increasing these would lead to even smaller test errors. However, we will concern ourselves more with properties of the SAMME algorithm than decision trees (where we know that increasing the depth will tend to decrease the test error). Therefore, we choose to cross-validate over the same range to see whether it is possible for a smaller number of classifiers may ever lead to a better test error.

In order to find the best performing combination, we performed 20 runs of 5-fold cross-validation on an 80% training data split, and selected the classifier with the greatest mean validation accuracy across the five folds. We then trained these classifiers on the full 80% training dataset, and tested them on the remaining 20% test dataset. The results are shown in Table 22.

In each run of cross-validation, 100 classifiers and a max depth of 8 was selected. Table 23 shows that these results have a mean test error of 3.60.

The fact that the largest number of classifiers performed best on test data in each experiment, rather than overfitting in comparison to smaller number of classifiers, is typical of boosting. Unlike for many algorithms, the test error tends to continue to decrease for a long time, even as the training error decreases to approach zero, as the margin between opposing classes tends to improve.

Number of classifiers	Max depth	Test error
100	8	3.98
100	8	3.39
100	8	4.14
100	8	2.74
100	8	3.39
100	8	3.01
100	8	5.00
100	8	3.44
100	8	3.55
100	8	3.60
100	8	3.23
100	8	3.60
100	8	3.76
100	8	4.41
100	8	2.53
100	8	3.55
100	8	3.17
100	8	3.39
100	8	3.66
100	8	4.46

Table 22: The best number of classifiers and maximum depth after performing 5-fold cross-validation on an 80% training data split, and its corresponding test error on the remaining 20% test data split, over 20 runs

Mean test error	Mean number of classifiers	Mean max depth
3.60 ± 0.587	100 ± 0	8 ± 0

Table 23: The mean of the best number of classifiers, maximum depth, and test errors

7 Extended discussion

(i) Comparison

Complexity In Section 1, we discussed and compared the complexity of the one-versus-rest and one-versus-one approaches. One-versus-rest trained $O(k)$ classifiers, while one-versus-one trained $O(k^2)$ which may be smaller in terms of the number of training examples they use, where k is the number of classes. For kernel algorithms, such as our implementations of support vector machines, which are between quadratic and cubic in the size of the training dataset depending on the value of C, one-versus-one may therefore be more efficient.

We have also seen that there is a training and test time complexity difference in using the primal or dual forms of algorithms. Letting m be the number of training examples, n be the number of features, and t the number of test examples:

- For training, the primal form explicitly stores weights corresponding to features of the training examples, which is costly if the number of features is large, such as when using kernel methods. The dual form stores coefficients associated with each training example, and is costly when the number of training examples is large, mainly due to the computation of the kernel Gram matrix over the training examples. However, the dual form's lack of dependence on the number of features makes it more computationally tractable to use kernels, which enables non-linear classification. For our kernel perceptrons, the training time is $O(m^2 n)$, due to the $O(m^2)$ Gram matrix computation.
- At test time, the primal form for prediction simply requires a dot product between a weight vector and the test example, so is $O(n)$ where n is the number of features, or $O(nt)$ for t test examples. The dual

form requires computing a kernel matrix between the training and test examples, and is $O(mnt)$ for t test examples.

We can multiply the complexity of one-versus-rest and one-versus-one by the kernel perceptron training and test complexities to get the total time complexity for each approach to multi-class generalization.

As discussed in Section 5, the training time complexity of each binary SVM varies between $O(m^2)$ and $O(m^3)$, depending on the number of support vectors, determined by the value of C. We generalize to k classes using one-versus-rest, so multiply by $O(k)$ to get $O(km^2)$ to $O(km^3)$. For prediction, the SVM complexity is proportional to the number of support vectors m_{SV}, with $O(m_{SV}n)$ for each test example.

Similarly, Section 6 discussed the complexity of the SAMME algorithm for multi-class AdaBoost. Using consistent notation here, the training complexity using decision trees as the underlying algorithm is $O(dnmlog(m)M)$, where d is the maximum depth, and M is the number of classifiers. This is more efficient than one-versus-rest which are $O(dnmlog(m)MK)$ and $O(dnmlog(m)MK^2)$. For prediction, the complexity is $O(log(m)M)$ per training example.

Performance Table 24 shows the results of the cross-validation sections for each algorithm in a single table. The best performing algorithm that we found was the one-versus-rest Gaussian kernel SVM with SMO, which had a mean test error of 2.12, followed by the CVXOPT approach, itself closely followed by the one-versus-rest polynomial kernel perceptron.

It seems plausible that the SAMME approach to multi-class AdaBoost could achieve even greater performance using the *SAMME.R* modification, which uses probability estimates rather than classifications to update the model, in addition to increasing the number of weak learners, and experimenting with the maximum tree depths.

An shown in Section 5, an SVM tries to maximize the margin between two classes of data, and the *soft-margin* form allows for some amount of training example misclassification, improving its generalization to test data. On the other hand, the perceptron treats the data as separable, and aims to find any hyperplane which separates the two classes, without optimizing a distance between the nearest opposing examples. It seems likely that this explains the performance difference between the perceptrons and SVMs.

Our cross-validation experiments showed that the polynomial kernel performed better than the Gaussian kernel perceptron. However, our results for both Gaussian kernel perceptrons and SVMs showed that the the Gaussian kernel width is a very sensitive hyperparameter, and it seems plausible that we did not choose optimal values to achieve the best performance. As further evidence, we found that using width 0.01 gave a mean test error of 3.15 over 20 runs in our basic results for the Gaussian kernel perceptron, but we cross-validated over values close to, but not exactly, 0.01.

Algorithm	**Mean test error**
Polynomial kernel perceptron (one-versus-rest)	3.23 ± 0.468
Gaussian kernel perceptron (one-versus-rest)	4.14 ± 0.567
Polynomial kernel perceptron (one-versus-one)	3.70 ± 0.860
Gaussian kernel SVM (one-versus-rest) with CVXOPT	2.89 ± 0.267
Gaussian kernel SVM (one-versus-rest) with SMO	2.12 ± 0.254
Multi-class AdaBoost (SAMME)	3.60 ± 0.587

Table 24: The mean and standard deviation of the test errors for the best combination of hyperparameters found via cross-validation, for each algorithm

2 Part II: Sparse learning

1

The 'just a little bit' problem We have m patterns $\boldsymbol{x}_1, ..., \boldsymbol{x}_m$ sampled uniformly at random from $\{-1, 1\}^n$, with labels $y_i := \boldsymbol{x}_{i,1}$, i.e. the label of a pattern is the value of its first feature.

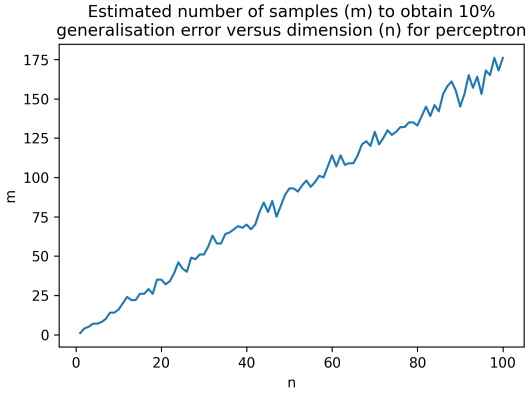
We are interested in estimating the sample complexity for this problem for a variety of algorithms, as a function of dimension (n), using a definition of sample complexity as the minimum number of examples (m) to incur no more than 10% generalization error (on average).

This is an example of a *sparse* learning problem, as our patterns are made up of many irrelevant features, since only the first feature matters for correct classification.

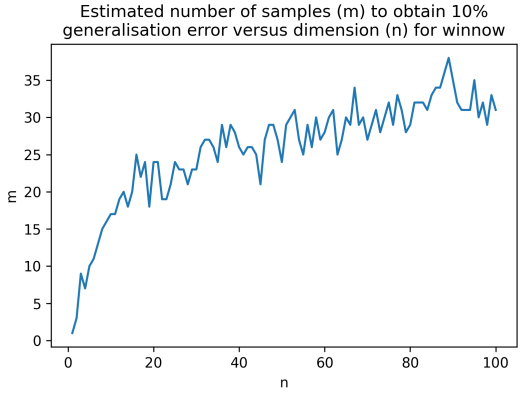
(i) Plots of estimated sample complexity

In Figure 9, we produce plots of the estimated number of samples to obtain 10% generalization error versus dimension for the perceptron, winnow, least squares, and 1-nearest neighbour algorithms.

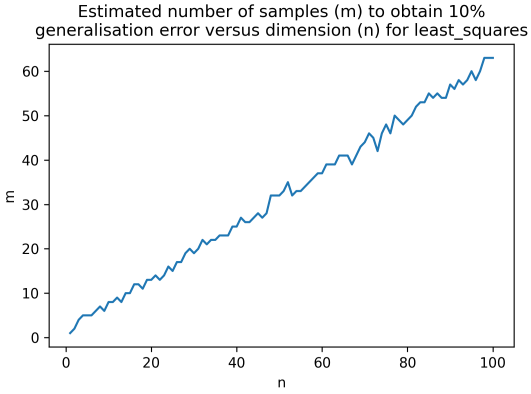
There are clear trends in each, with perceptron and least squares increasing linearly, winnow increasing logarithmically, and 1-nearest neighbour exponentially.



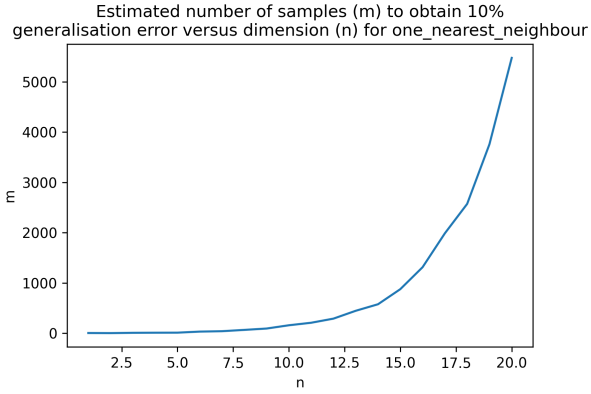
(a) Perceptron



(b) Winnow



(c) Least squares



(d) 1-nearest neighbour

Figure 9: Estimated number of samples (m) to obtain 10% generalization error versus dimension (n)

(ii) Method for estimating sample complexity

We let $\mathcal{S}_m$ denote a set of m patterns drawn uniformly at random from $\{-1, 1\}^n$ with their derived labels. Then let $\mathcal{A}_{\mathcal{S}}(\boldsymbol{x})$ denote the prediction of an algorithm $\mathcal{A}$ trained from data sequence $\mathcal{S}$ on pattern $\boldsymbol{x}$. The

generalization error is then

$$\mathcal{E}(\mathcal{A}_{\mathcal{S}}) := 2^{-n} \sum_{\boldsymbol{x} \in \{-1,1\}^n} I[\mathcal{A}_{\mathcal{S}}(\boldsymbol{x}) \neq x_1]. \tag{7}$$

Thus, the sample complexity on average at 10% generalization error is

$$C(\mathcal{A}) := \min\{m \in \{1, 2, ...\} : \mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})] \leq 0.1\}. \tag{8}$$

As the dimension (n) increases, this becomes increasingly computationally expensive to compute exactly, with the speed at which it becomes so depending on the underlying algorithm being trained.

For the 'just a little problem', there are 2^n possible distinct samples for each dimension n, so it is computationally infeasible to compute the exact generalization error by using Equation 7 with a test dataset which contains *every* sample in the space, in order to get the sample complexity for increasing n.

Instead, for each dimension n and number of patterns m, we will test each algorithm on randomly generated subsets of the sample space. We will compute $\mathcal{E}(\mathcal{A}_{\mathcal{S}})$ over a number of runs with such subsets, and take an average in order to estimate the expected generalization error $\mathbb{E}[\mathcal{E}(\mathcal{A}_{\mathcal{S}_m})]$ in Equation 8.

We implement two methods to estimate the sample complexity. The plots in Figure 9 used Method 1. In both methods, we compute the expected value over 10 runs, and with a test dataset sizes of $10,000$ (generated without replacement). For $n > 13$, this means the size of the space of possible samples is greater than the size of the test dataset, and this increases with 2^n. As we are looking for trends by estimating the sample complexity for n up to 100, we are estimating using a tiny subset of the space.

Method 1: Increasing from $m = 1$ until the minimum is found In our first method, we loop over every dimension n in order to estimate its sample complexity, using our definition in Equation 8. For each algorithm, we train on $m = 1$ examples, and compute the generalization error similar to Equation 7, but using a test dataset of size $10,000$. We do so 10 times, and take an average to get an estimate of the expected generalization error. We check if this is less than or equal to 0.1, in which case we stop, returning m for that particular n. If it is greater than 0.1, we increment m, and repeat each step. For a particular n, this method is $O(m)$ in m.

Method 2: Finding the minimum via binary search In our second method, we use a binary search approach to narrow down the m which is the minimum number of examples to incur no more than 10% generalization error on average. We again loop over each dimension n to find the minimum m for each. In each loop, we initialize the lower and upper bounds within which to search for the minimum m, which can be specified by the user based on their algorithm. The generalization error is computed as before for $m = \text{floor}((L+R)/2)$. If it is less than 0.1, the upper bound is set to m, as we are interested in the minimum m which has this property. The minimum such m with this property is tracked. If it is greater than m, the lower bound is set to m, as the minimum m must be smaller than this. When the loop converges, the tracked minimum m is the estimate for the given n. For a particular n, this method is $O(\log m)$ in m.

It is possible that the specified lower and upper bounds may not contain the number of samples m which gives a generalization error. However, with experience through Method 1, we found this was not a problem.

We found that this method was particularly effective for estimating the sample complexity of 1-nearest neighbour (using lower and upper bounds of 1 and $10,000$ between dimensions 1 and 20), at least for the range of dimensions that we estimated, speeding up the computation to some number of minutes from hours. It also sped up the computation for the other methods, although these were relatively quick for the dimensions we chose.

It seems reasonable a priori, and particularly after gathering our experimental evidence, to assume that the sample complexity is an increasing function of n for our algorithms, so we could choose to speed up Method 1 by setting the starting value of m for the n^{th} loop to be the minimum m for the $(n-1)^{th}$ loop. Similarly, in Method 2, the lower bound on m for the n^{th} loop could be set to the minimum for the $(n-1)^{th}$ loop. In practice, the computation was efficient enough without this for our dimension range.

Our perceptron, winnow, and least squares algorithms all learn hypotheses in the set of halfspaces, for which the VC dimension is n, while it is infinite for nearest neighbours.

In the context of empirical risk minimization, we can decompose the risk of a hypothesis $h_S = \mathrm{ERM}_H(S)$ in space H as

$$L_{\mathcal{D}}(h_S) = \epsilon_{\text{app}} + \epsilon_{\text{est}},$$

where $\epsilon_{\text{app}} = \min_{h \in H} L_{\mathcal{D}}(h)$ is the approximation error, and $\epsilon_{\text{est}} = L_{\mathcal{D}}(h_S) - \min_{h \in H} L_{\mathcal{D}}(h)$ is the estimation error.

The estimation error decreases as the number of samples (m) increases, while it increases as the complexity of the hypothesis space, H, increases. In our case, it grows according to 2^n, where n is the dimension.

For the perceptron, winnow, and least squares algorithms, the approximation error is zero for our problem, since they are all able to learn the weight vector which puts all the weight on the first feature, so $L_{\mathcal{D}}(h_S) = \epsilon_{\text{est}}$, i.e. the risk, or generalization error, is given by the training error for a sample. By repeating the training process with many samples and taking a mean as in Method 1 and 2, we therefore reduce the variance in the estimate of the generalization error.

(iii) Sample complexity estimates and comparison

The plots in Figure 9 seem to show the sample complexity for perceptron and least squares increasing linearly, winnow increasing logarithmically, and 1-nearest neighbour exponentially.

In Figure 10, we have fitted curves according to these assumption and plotted them against the estimates generated by Method 1, which seem to match up. The fitted equations were:

- Perceptron: $0.265 + 1.736 \cdot n$
- Winnow: $0.004 + 7.097 \cdot \log(n)$
- Least squares: $1.03 + 0.61 \cdot n$
- 1-nearest neighbours: $0.926 \cdot \exp(0.391 \cdot n)$

Therefore, experimentally from the plots, it seems that the sample complexity of each algorithm grows according to Table 25, where we have used Θ as it seems that the estimate of the sample complexity is bounded above and below by a constant multiplied by the fit.

Algorithm	**Complexity**
Perceptron	$\Theta(n)$
Winnow	$\Theta(\log n)$
Least squares	$\Theta(n)$
1-nearest neighbours	$\Theta(\exp n)$

Table 25: The sample complexity for each algorithm, that we predict experimentally from the plots in Figure 10

For our sparse learning problem, where all but 1 of our features are irrelevant, these results are as we may expect.

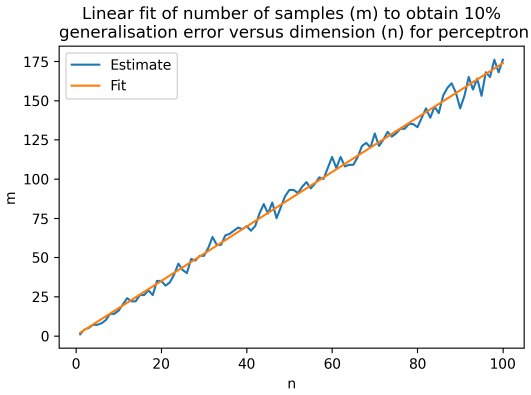
The winnow algorithm was introduced as a perceptron-like algorithm that used a multiplicative weight-update scheme [6] to achieve a bound that is logarithmic in the number of mistakes as a function of the number of irrelevant features, while the perceptron bound is linear.

For learning the best k-literal monotone disjunction over n variables, the mistake bound on the perceptron is known to be

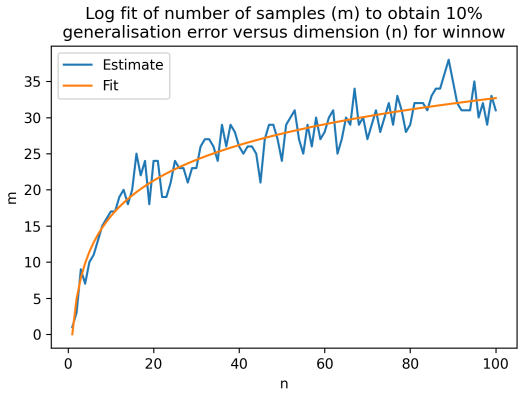
$$M \leq (4k + 1)(n + 1),$$

which is linear, or $\Theta(n)$, in the number of dimensions, consistent with our experimental observations. Similarly, for the winnow algorithm, the mistake bound is

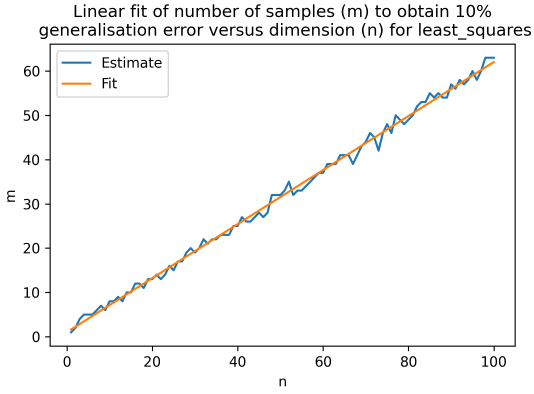
$$M \leq 3k(\log n + 1) + 2,$$



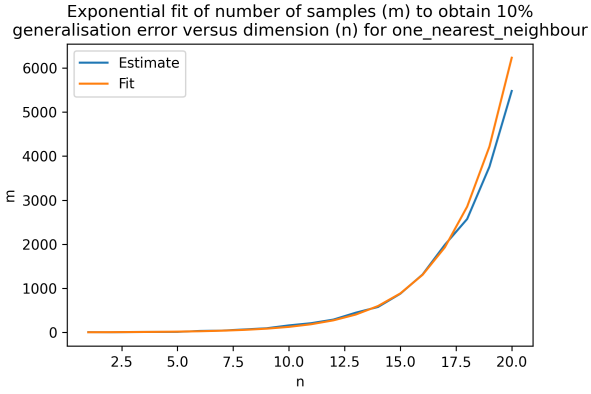
(a) Perceptron



(b) Winnow



(c) Least squares



(d) 1-nearest neighbour

Figure 10: Fits of the estimated number of samples (m) to obtain 10% generalization error versus dimension (n)

which is also consistent with our observation of $\Theta(\log n)$ in the number of dimensions.

A priori, we expect a perceptron to perform better for tasks where the number of important features is larger, while the samples are sparse, and vice-versa for the winnow algorithm. The perceptron struggles more than winnow to find a separating hyperplane when many features are irrelevant.

We can compare the weights after training and estimated sample complexities, as in Table 26, to see this further. The table shows the sample complexity estimate over a single run, for the samples with 100 features, and a sample of the first ten weights on those features, where weight index 1 is the relevant feature. We can see that (as the plots in Figure 9 made clear) the winnow requires many fewer samples to discover the important feature, and the weights on other features also seem smaller.

On a mistake, the winnow algorithm is able to double the weight on the first index, while the perceptron may only update it by a maximum of 1, and it may update irrelevant weights in an unhelpful direction, too. Intuitively, it makes sense that it will take more time for the irrelevant weights to balance out as the number of samples, m, increases, for the perceptron.

Algorithm	**Dimension (n)**	**Estimate (m)**	**Weight index**									
			1	2	3	4	5	6	7	8	9	10
Perceptron	100	183	40	-2	-4	-4	2	0	0	-2	-4	6
Winnow	100	29	64	0.5	0.25	2	0.125	0.5	4	0.5	1	0.5
Least squares	100	61	0.726	-0.00510	-0.0303	0.0311	-0.0588	0.0417	0.00571	0.0312	0.00711	0.0611

Table 26: Estimated sample complexity (m) for dimension $n = 100$, and their associated weights, over a single run

The least squares algorithm finds the weights which minimize the squared loss,

$$\arg\min_{w} \frac{1}{m} \sum_{i=1}^{m} (\langle w, x_i \rangle - y_i)^2,$$

through a single matrix inversion. Intuitively, least squares outputs $\langle w, x_i \rangle$ and is adjusted using the square of the difference between this and the label, so it is adjusted even if the sign of its output is correct, which seems helpful when all but one feature is irrelevant.

Table 26 is also suggestive that in comparison to the perceptron, least squares needs fewer examples, despite the fact that their sample complexities both seem linear, or $\Theta(n)$ in the dimension n, so the constant in the big O estimate would be smaller.

Finally, the estimated exponential sample complexity of 1-nearest neighbours for our problem also coincides with our knowledge of nearest neighbours algorithms. The number of possible examples is exponential in the dimension n, at 2^n. It is computationally infeasible to store anywhere close to each possible example, but this means that the examples will be far away from each other on average.

As only the first feature is relevant, this problem is exacerbated, as for 1-nearest neighbours to predict correctly, a training example must share in the first feature of a test example, while being nearest overall when including the many irrelevant features. Intuitively, this means the sample complexity m will also need to grow exponentially in n, like the space of possible samples is 2^n. This is a handicap that the other algorithms do not share to this extent: they can more easily down-weight the irrelevant features.

(iv) Upper bound on perceptron mistake probability

Suppose $s \in \{1, ..., m\}$ is sampled uniformly at random. We can derive an upper bound $\hat{p}_{m,n}$ on the probability that a perceptron will make a mistake on the sth example after being trained on $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_{s-1}, y_{s-1})$, for the 'just a little bit' problem.

By Novikoff's Theorem, we know that for the sequences of examples $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_s, y_s)$, the number of mistakes of the Perceptron algorithm is upper bounded by

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

where $R := \max_t ||\boldsymbol{x}_t||$.

For our problem, we know that $R = \max_t \sqrt{\sum_{i=1}^{n} \boldsymbol{x}_{t,i}^2} = \sqrt{n}$, where n is the dimension of the samples, as the patterns $\boldsymbol{x}_t$ are drawn from $\{-1, 1\}^n$. Hence, the mistake bound is given by

$$M \leq \frac{n}{\gamma^2}.$$

Setting $B := \frac{n}{\gamma^2}$, we can say that an upper bound on the probability of making a mistake on the sth example is

$$\hat{p}_{m,n} \leq \frac{n}{m\gamma^2} = \frac{B}{m}, \tag{9}$$

since there are no more than B trials with mistakes, and s is drawn uniformly.

References

[1] Léon Bottou and Chih-Jen Lin. "Support vector machine solvers". In: *Large scale kernel machines* 3.1 (2007), pp. 301–320.

[2] Wiesław Chmielnicki and Katarzyna Stąpor. "Using the one–versus–rest strategy with samples balancing to improve pairwise coupling classification". In: *International Journal of Applied Mathematics and Computer Science* 26.1 (2016), pp. 191–201.

[3] Koby Crammer and Yoram Singer. "On the algorithmic implementation of multiclass kernel-based vector machines". In: *Journal of machine learning research* 2.Dec (2001), pp. 265–292.

[4] Trevor Hastie et al. "Multi-class adaboost". In: *Statistics and its Interface* 2.3 (2009), pp. 349–360.

[5] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[6] Nick Littlestone. "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm". In: *Machine learning* 2.4 (1988), pp. 285–318.

[7] John Platt. "Sequential minimal optimization: A fast algorithm for training support vector machines". In: *MSR-TR-98-14* (1998).

[8] "The Simplified SMO Algorithm". In: *CS 229, Autumn 2009, Stanford University* (2009).