Mathematical Aspects
of Machine Learning

Report

Digit Recognition with Support Vector Machines

Authors:
Lisa Gaedke-Merzhäuser
Paul Korsmeier
Lisa Mattrisch
Vanessa Schreck

1 Problem Statement and Introduction to SVM

The main goal of our project was to correctly identify handwritten digits based on the MNIST ("Modified National Institute of Standards and Technology") data set. This data set consists of 42,000 gray-scale images. Each image is 28 pixels in height and 28 pixels in width. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel (kag).

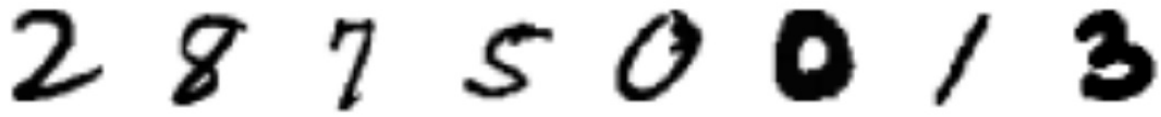

Figure 1: Visualization of eight of these images

Formally speaking, we have points $\{x_i\}_{i=1}^{l} \subset \mathbb{R}^{28^2}$ with respective labels from $\{0, 1, \ldots, 9\}$. Our goal is to now find a classification function f that gives for each of these points a prediction $f(x) \in \{0, 1, \ldots, 9\}$ which ideally matches the actual label.

There are several ways to achieve this goal, one of which is the concept of kernelized support vector machines (SVM). The main idea behind this algorithm is to not only separate the data according to their labels, but also to do so in a way that is as reasonable as possible, i.e. we want to maximize the distance of the data to the decision boundary. We first transfer the data with a feature map Φ into a suitable feature space and look for a linear separation there.

From this notion, two particular problems arise. Firstly how to solve the optimization problem of maximizing the margin (cf. Section 1.1 and 2) and secondly: The SVM is a binary classifier, i.e. it can only handle data that is divided into two classes. We therefore decided to compare different ways to deal with this obstacle, generalizing the application of SVMs to multiclass classification problems (cf. Section 3). The methods that we considered all combine several instances of the SVM in its original form, classifying according to labels $\{\pm 1\}$.

We can summarize this binary classification problem in the objective of finding a decision function of the form

$$f(x) = \operatorname{sgn}\left(w^T \Phi(x) + b\right).$$

1.1 Binary Support Vector Machine

Approaching binary support vector machines from a more technical side, we find that they aim to find the maximum margin hyperplane in the feature space, i.e. the goal is to maximize the margin while softly penalizing points that lie on the wrong side of the boundary or inside the margin (Bis06). Dualizing this optimization problem, we obtain the following equivalent quadratic program:

$$\begin{aligned} \text{minimize} \quad & d(\alpha) := \tfrac{1}{2}\alpha^T Q \alpha - 1^T \alpha \\ \text{s.t.} \quad & 0 \leq \alpha \leq C \quad \text{and} \quad y^T \alpha = 0, \end{aligned} \tag{1.1}$$

where α is the dual variable, $Q_{ij} = y_i y_j k(x_i, x_j)$, k the kernel function and $C = 1/(2\lambda l)$ for the penalty term λ of the soft margin SVM. After finding the optimal solution α^* of this program, we can formulate the resulting decision function as

$$f(x) = \operatorname{sgn}\left(\sum_{i=1}^{l} \alpha_i^* y_i k(x_i, x_j) + b\right).$$

Note that only data points x_i with $\alpha_i \neq 0$ appear in the decision function. Those points are called *support vectors*.
By analyzing primal, dual and the corresponding KKT conditions, we find that any x_i with $\alpha_i = C$ was correctly classified and lies exactly on the margin boundary, and any x_i with $0 < \alpha_i < C$ lies either inside the margin or on the wrong side of the margin. Additionally, any data points x_i on the (correct) margin boundary must satisfy $y_i f(x_i) = 1$. We can use this property to find the parameter b (Bis06). Hence, if we succeed to find an optimal solution to (1.1), we have everything we need to construct the decision function.

2 Solving the optimization problem with SMO

2.1 Some theory on SMO convergence

Let α be some feasible variable for Problem (1.1). Defining

$$\mathcal{L}(\alpha, \delta, \mu, \beta) := \frac{1}{2}\alpha^T Q \alpha - 1^T \alpha - \delta^T \alpha + \mu^T(\alpha - C) - \beta \alpha^T y$$

$$F_i(\alpha) := y_i(\partial_i d)(\alpha) = \sum_{i=1}^{l} \alpha_j y_j k(x_i, x_j) - y_i \quad \text{for} \quad i = 1, \ldots, l,$$

by careful manipulations we find that the KKT optimality conditions

$$\left.\begin{aligned} \nabla_\alpha \mathcal{L}(\alpha^*, \delta^*, \mu^*, \beta^*) &= 0 \\ \delta_i^* &\geq 0 \\ \delta_i^* \alpha_i^* &= 0 \\ \mu_i^* &\geq 0 \\ \mu_i^*(\alpha_i^* - C) &= 0 \\ \alpha_i^* &\text{ feasible} \end{aligned}\right\} \text{for all } i \in \{1, \ldots, l\}$$

for a solution of Problem (1.1) (which are sufficient, since Q is spsd), are equivalent to the – much simpler looking – pairwise condition

$$b_{up}(\alpha) := \min_{i \in I_{up}(\alpha)} F_i(\alpha) \geq \max_{j \in I_{low}(\alpha)} F_j(\alpha) =: b_{low}(\alpha),$$

where $I_{up}(\alpha)$ and $I_{low}(\alpha)$ are subsets of the index set $\{1, \ldots, l\}$ defined by

$$I_{up}(\alpha) := \{i \mid \alpha_i < C \text{ and } y_i = 1 \text{ or } \alpha_i > 0 \text{ and } y_i = -1\}$$
$$I_{low}(\alpha) := \{j \mid \alpha_j < C \text{ and } y_j = -1 \text{ or } \alpha_j > 0 \text{ and } y_j = 1\}.$$

Any pair $(i, j) \in I_{up}(\alpha) \times I_{low}(\alpha)$ with $F_i(\alpha) < F_j(\alpha)$ is thus called a *violating pair* and an objective equivalent to solving Problem (1.1) is to change α so as to remove all such

violating pairs. Since a priori we do not know if the solution α^* fulfils (2.1) strictly or not, we define, for some small tolerance $\tau > 0$, a τ-violating pair as some $(i, j) \in I_{up}(\alpha) \times I_{low}(\alpha)$ which satisfies $F_i(\alpha) < F_j(\alpha) - \tau$ and require that all τ-violating pairs be removed, or equivalently,

$$b_{up}(\alpha) \geq b_{low}(\alpha) - \tau.$$

It holds (see (KG02) for a proof) that any algorithm of the following form terminates after finitely many steps:

Algorithm 1 (General SMO type algorithm)**.** Let $\tau > 0$. Initialize $k = 0$ and $\alpha^0 = 0$ and generate iterates α^k, $k \in \mathbb{N}$, as follows:

1. If α^k satisfies (2.1), stop. Else choose a τ-violating pair $(i, j) \in I_{up}(\alpha^k) \times I_{low}(\alpha^k)$.
2. Minimize d varying only α_i^k and α_j^k, leaving α_n^k fixed for $n \notin \{i, j\}$ and respecting the constraints of Problem (1.1) to obtain α^{new}.
3. Set $k := k + 1$, $\alpha^k := \alpha^{\text{new}}$ and go to Step 1.

Platt's original Sequential Minimal Optimization (SMO) algorithm (actually, we are referring to "Modification 1" by (KSBM01)), although ground-breaking, has two weaknesses. Firstly, its pseudocode description is quite complex, making it hard to judge whether or not one has implemented it as intended by its originators. Secondly, by trying to avoid computational effort in the *while* steps, it actually runs much longer than all other algorithms we have implemented or tested.

In short, Platt's SMO tries first to ensure that

$$b_{up,I_0}(\alpha) := \max_{i \in I_0(\alpha)} F_i(\alpha) \geq \min_{j \in I_0} F_j(\alpha) =: b_{low,I_0}(\alpha),$$

where $I_0(\alpha) := \{i \mid 0 < \alpha_i < C\}$ is the index set of "interior" α_is. A cache of only the F_i for $i \in I_0$ is kept until this is achieved. Then all $i \in \{0, \dots l\}$ are examined, and the cache of correctly stored F_is (along with the indices $\widetilde{i_{up}}$ and $\widetilde{i_{low}}$ indicating where the so far most extreme F_i occur) is extended as long as an α_j is found such that $(j, \widetilde{i_{low}})$ or $(\widetilde{i_{up}}, j)$ is violating, depending on whether $j \in I_{up}$ or $j \in I_{low}$.

Therefore, the algorithm does not resolve a *maximally* violating pair, by which we mean some

$$(i_{up}, i_{low}) \in \operatorname{argmin}_{i \in I_{up}(\alpha)} F_i(\alpha) \times \operatorname{argmax}_{j \in I_{low}(\alpha)} F_j(\alpha).$$

To look up such a pair is the strategy of the "Working Set Selection 1" (WSS1) approach in (FCL05) and a rewarding investment, see benchmarking section.

Parametrize the 1D Problem in Step 2 of Algorithm 1 as

$$\alpha_i(t^*) = \alpha_i^k + y_i t*, \quad \alpha_j(t^*) = \alpha_j^k - y_j t*, \quad \alpha_n(t^*) = \alpha_n^k \text{ for } k \notin \{i, j\}$$

for some optimal feasible $t^* \in \mathbb{R}$ and set $\Phi(t) = d(\alpha(t))$ as the 1D objective function. Then actually

$$F_i(\alpha) - F_j(\alpha) = \Phi'(0),$$

so WSS1 corresponds to a steepest descent approach aiming for a substantial decrease in d. Since this uses only first order information, a second "Working Set Selection 2" (WSS2)

strategy is proposed in (FCL05), which (at relatively low extra cost) employs second order information to choose a violating pair almost optimally: $i \in \operatorname{argmin}_{i \in I_{up}(\alpha)} F_i(\alpha)$ and $j \in I_{low}$ such that the decrease in d produced by removing the violating pair is maximal.

2.2 Implementational issues

One particular source of trouble was that WSS1 and WSS2 sometimes got stuck on a single violating pair, which is extremely puzzling because – in theory – a violating pair (i, j) of indices can no longer be violating after an SMO step has been taken. It turned out that the precise definitions of I_{up} and I_{low} had to be softened so that numerical errors do not stop the algorithm from recognizing that $(\alpha_i^{new}, \alpha_j^{new}$ has hit the boundary of $[0, C]^2$. This is crucial for steps where the box constraints are active for $(\alpha_i^{new}, \alpha_j^{new})$. For further details, please see the commented code we provided.

2.3 Benchmarking different SMOs

We implemented Platt's SMO, WSS1 in two variations and WSS2. The first variation of WSS1 caches all computations of Gramian matrix rows in order to avoid expensive double computations. The other variant of WSS1 is forgetful. Taking the scikit-learn SVC algorithm as a 5th (tough) competitor, we tested the algorithms at hand for the first classifier of ECOC for a range of numbers of MNIST training points.

As can be seen, the speediest algorithms are, in decreasing order: scikit-learn, WSS1 with caching, WSS2 with caching, WSS1 without caching, Platt's SMO. Somewhat surprisingly, WSS2 with caching is not the fastest method, although in theory it should be superior to WSS1 with caching. Apparently, the extra computational cost outweighs the theoretical gain. We suppose that the strange kink in the WSS1 with caching graph (Figure 3) is due to multicore features of Python kicking in at 1800 training points. The benchmarks were carried out on a Windows 7 x64 machine with an Intel i5 quad core processor at 4 GHz and 16 GB of RAM.

The polynomial order of all algorithms is 2, judging from the loglog plot in Figure 4 and a linear least squares fit of the loglog data for WSS1 with caching. The latter also yielded that our WSS1 with caching takes about 5.8 times as long as scikit-learn.

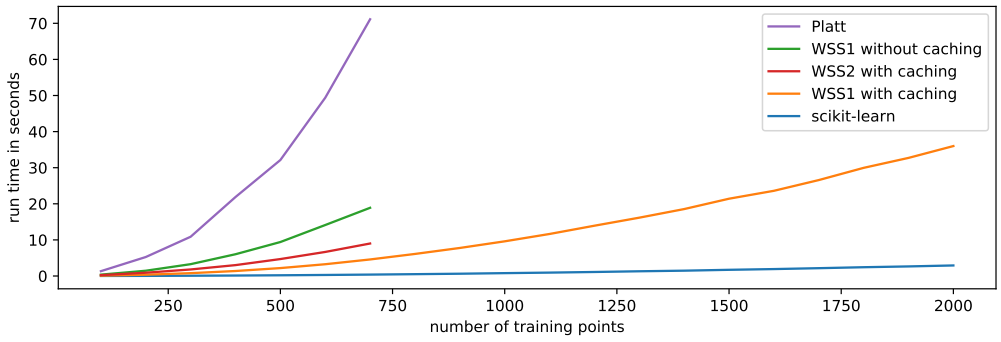

Figure 2: Run times for different SMO algorithms, Gaussian kernel

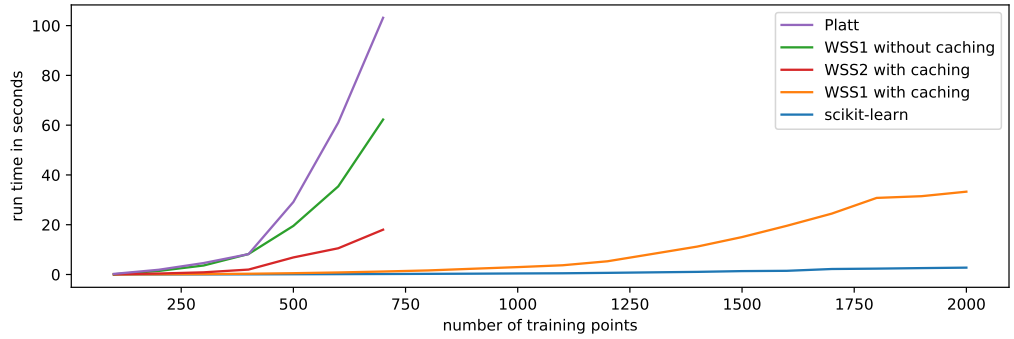

Figure 3: Run times for different SMO algorithms, standard scalar product

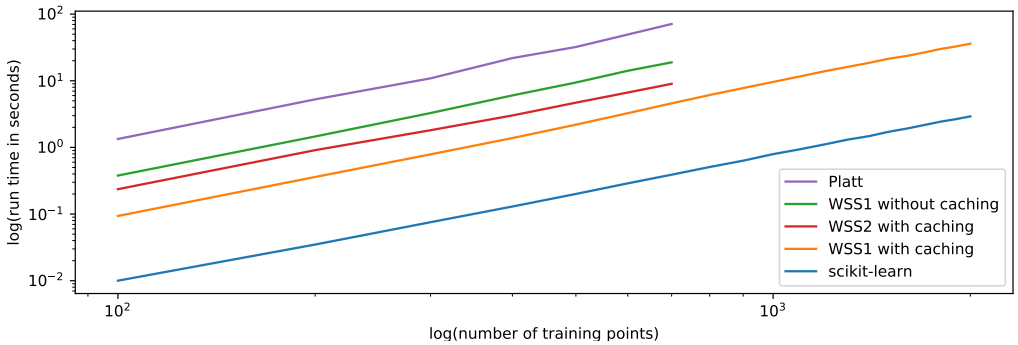

Figure 4: loglog plot of run times, Gaussian kernel

3 Multiclass Classification Problem

As mentioned in the beginning Support Vector Machines are binary classifiers with two possible output labels. Often one chooses them to be 1 and -1. Our problem however was to classify unknown data into one out of ten different categories. Since there is no direct way to change one support vector machine so that it can choose from more than two labels we had to find a way to combine several SVMs, who together would be able to do so. There are a number of well known strategies tackling this issue. Let us first look at a very intuitive but also primitive approach called One-vs-All classification. Using this method one trains as many SVMs as there are classes, so ten in our case, and sets labels such that the i-th SVM has all training data with label i set to 1 and everything else to -1. When trying to predict a new label, one simply looks for a classifier that gave this data point the label 1. Unfortunately, this approach has some issues. First of all, new data points are generally not uniquely classified. Therefore, it is necessary to add some sort of confidence score that signals which label is considered to be the most likely out of the possible ones. Finding good ways of assigning reliable confidence scores is often not an easy task. We decided to resolve this issue by computing the barycenters of each class, and when ambiguities arose, we chose the label of the closest barycenter out of the positive labels. This simple strategy improved our testing results. We decided on not devoting any more time on implementing a different strategy to determine confidence scores because the One-vs-All classification has another major drawback. The number of training points for each classifier being assigned positive and negative labels is very unbalanced, in our case on average 1 to 9. Therefore, one is likely to get shifted margins

to what one would normally perceive to be correct. Since the negative class is simply greater in size, the penalty term will draw more attention towards avoiding points with negative labels inside the margin and henceforth neglecting the positive labels.

Another common intuitive approach would be the One-vs-One classification. Here, during training one only ever considers the data corresponding to two classes and sets one label to 1 and the other one to -1 for every possible pair. In our case this would mean training $\binom{10}{2} = 45$ different SVMs however with data sets of a fifth of the original size. Because in classification algorithm the issue of how to deal with ambiguities gets even worse, we decided not to implement this approach and instead focus our attention on a different strategy that seemed like a much better idea to us. And indeed, as the last chapter will reveal, this yielded very good results.

3.1 Error-Correcting Output Codes

Error-correcting output codes (ECOC) are easiest to explain considering a concrete example. We trained 15 different classifiers f_i. The rows of the table show what class is assigned which label depending on each classifier. For example f_0 assigns 1 to all even numbers and -1 to all odd numbers. This way, each class has a string or code word consisting of 1's and -1's it corresponds to.

Table 1: Error Correcting Output Codes

Class	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}
0	1	1	-1	-1	-1	-1	1	-1	1	-1	-1	1	1	-1	1
1	-1	-1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	-1
2	1	-1	-1	1	-1	-1	-1	1	1	1	1	-1	1	-1	1
3	-1	-1	1	1	-1	1	1	1	-1	-1	-1	-1	1	-1	1
4	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1
5	-1	1	-1	-1	1	1	-1	-1	1	1	-1	-1	-1	-1	1
6	1	-1	1	1	1	-1	-1	-1	-1	1	-1	1	-1	-1	1
7	-1	-1	-1	1	1	1	1	-1	1	-1	1	1	-1	-1	1
8	1	1	-1	1	-1	1	1	-1	-1	1	-1	-1	-1	1	1
9	-1	1	1	1	-1	-1	-1	-1	1	-1	1	-1	-1	1	1

The code words are chosen such that their Hamming distance (i.e. the number of entries where they differ) is maximized. In our case, each code word has a Hamming distance of at least seven to any of the others strings. Now, when one wants to label an unknown data point, each of the classifiers assigns it a label and one eventually gets an output string of 15 digits. We classify the data point according the code word it has the least Hamming distance to. The name *Error Correcting Output Codes* is derived from the fact that for example any three classifiers can misclassify a data point and the algorithm still outputs the correct result. Depending on the difficulty of the problem one could choose more or less classifiers and hece in- or decrease the Hamming distance of the set of code words. We took this idea as well as the code words given above from (DB95).

4 Results, Difficulties & Conclusions

This is the largest programming project any one of has ever worked on and neither of us had done much with Python before or had any other machine learning experience. So at any given time there was always a great variety of difficulties at hand. And because we implemented everything ourselves the problem could be literally anywhere, from misunderstanding the pseudocode we used, over float and integer division or git merging issues to immense troubles with the internal solvers for the optimization problem. But because of that every little success was a source of joy which grew with every working block of code we managed to compile.

The overall structure of our program looks as follows. We wrote our own `mySVMclass`, which is a self contained object with a number of attributes and callable functions, which includes the `SMOsolver` and `crossValidation`. We then have separate .ipynb notebooks for the two classification algorithms where we specify training data sets, the kernel, the different parameters and create instances of our `mySVMclass`. Especially in the case of the gaussian kernel finding suitable parameters was very essential. We therefore wrote a separate notebook where we systematically test them and determine which ones seem optimal. We also automatically save our trained SVMs for the different sized training data in binary format. We also visualized toy data sets in lower dimensions to better understand how our algorithms behave in different cases.

Table 2: **Overview Results of Correctly Classified Digits**

# training points	One-vs-All uniquely classfied, linear	One-vs-All with bary-centers, linear	One-vs-All uniquely classfied, Gaussian	One-vs-All with bary-centers, Gaussian	ECOC, linear	ECOC, Gaussian
500	65.9%	74.1%	75.4%	83.3%	74.2%	87.4%
1000	68.2%	75.0%	84.3%	89.0%	78.0%	92.7%
2000	70.2%	76.4%	89.8%	91.9%	77.8%	94.3%
5000	70.0%	73.8%	88.9%	91.6%	82.0%	95.2%
10000	64.6%	67.5%	88.0%	90.6%	82.5%	95.4%

Our given data set included 42,000 handwritten digits. We never used all of them for training because this large amount of training data simply exceeded our available compute power. But from the table below we can see that it would probably not have improved our results considerably or might even have worsened them. And this way we always had engouh labeled data to test and verify our results with. In the table you can find the results of the different multi-class classifiers. For the One-vs-All approach we give numbers for how many labels were found correctly with and without using additional classification by location of the barycenters, so that one can really see how many uniquely classified data points are labeled correctly. The percentages are derived from always testing with 1,000 data points that were of course not used for training beforehand. Unsurprisingly ECOC had longer run times than the One-vs-All classifier, it has to train 15 instead of only 10 and the computation with the Gaussian kernel took longer than with the linear kernel, which was also to be expected because of its higher complexity (more detail?). For 10,000 training points ECOC with linear kernel took 1h 16 min and ECOC with gaussian

kernel took 2 h 26 min for training.

As expected from theoretical considerations the Error-Correcting Output Codes performed better than the One-vs-All classification for training sets of any size. However we were surprised how well One-vs-All with the gaussian kernel worked in the end. Every picture of a hand-written digit gets turned into a vector of size 784 ($= 28^2$), it was not clear that the output data would be anywhere close to being linearly separable so we were also suprised to see the linear classifiers working at all, especially in the first case were not a single ambiguity or mistake could be corrected. With an increase of training points the results first increase across the board. The performance of classifiers with linear kernel then often starts to decrease, sometimes drastically, with training sets of size 2,000. We assume that this phenomenon occurs because the more training data we have the harder it gets to linearly separate it.

We have learned a lot in the past 4 weeks and although this program does not (yet?) win us a kaggle competition we are in general (very) satisfied with the overall outcome of the project and especially with our success rate of over 95%.

References

[Bis06] Bishop, Christopher M.: *Pattern Recognition and Machine Learning* -. 1st ed. 2006. Corr. 2nd printing 2011. Berlin, Heidelberg : Springer, 2006. – ISBN 978–0–387–31073–2

[DB95] Dietterich, Thomas G. ; Bakiri, Ghulum: Solving multiclass learning problems via error-correcting output codes. In: *Journal of artificial intelligence research* 2 (1995), S. 263–286

[FCL05] Fan, Rong-En ; Chen, Pai-Hsuen ; Lin, Chih-Jen: Working Set Selection Using Second Order Information for Training Support Vector Machines. In: *J. Mach. Learn. Res.* 6 (2005), Dezember, S. 1889–1918. – ISSN 1532–4435

[kag] `https://www.kaggle.com/c/digit-recognizer/data`

[KG02] Keerthi, S.S. ; Gilbert, E.G.: Convergence of a Generalized SMO Algorithm for SVM Classifier Design. In: *Machine Learning* 46 (2002), Jan, Nr. 1, S. 351–360. – ISSN 1573–0565

[KSBM01] Keerthi, S. S. ; Shevade, S. K. ; Bhattacharyya, C. ; Murthy, K. R. K.: Improvements to Platt's SMO Algorithm for SVM Classifier Design. In: *Neural Computation* 13 (2001), Nr. 3, S. 637–649