APPENDIX

A SVM - APPLICATION EXAMPLES

A.1 Hand-written Digit Recognition

The first real-world task on which Support Vector machines were tested was the problem of hand-written character recognition. This is a problem currently used for benchmarking classifiers, originally motivated by the need of the US Postal Service to automate sorting mail using the hand-written ZIP codes. Different models of SVM have been tested on the freely available datasets of digits: USPS (United States Postal Service) and NIST (National Institute for Standard and Technology).

For USPS data, where the input space is 256 dimensional, the following polynomial and Gaussian kernel were used:

\[ K(x, y) = \left( \frac{\langle x \cdot y \rangle}{256} \right)^d \]

\[ K(x, y) = \exp\left(-\frac{\|x - y\|^2}{256\sigma^2}\right) \]

for different values of \( d \) and \( \sigma \).

For polynomial kernels, degrees from 1 to 6 have been tested, for Gaussian kernels, values of \( \sigma \) between 0.1 and 4.0. The USPS are reported to be totally separable with a maximal margin machine starting from degree 3 whereas lower values with the 1-norm and 2-norm approach generated errors.

This whole set of experiments is particularly interesting, because the data have been extensively studied and there are algorithms that have been designed specifically for this dataset. The fact that SVM can perform as well as these systems without including any detailed prior knowledge is certainly remarkable.
A.2 Text Categorization

The task of text categorization is the classification of natural text (or hypertext) documents into a fixed number of predefined categories based on their content. This problem arises in a number of different areas including email filtering, web searching, office automation, sorting documents by topic and classification of news agency stories. Since a document can be assigned to more than one category this is not a multiclass classification problem, but can be viewed as a series of binary classification problems, one for each category.

There are many resources in this field available in the internet, so we won’t go into detail here. But one interesting work should be noted here which also led to a library for SVMs with its own algorithm:

The text categorization of the Reuters’ News from Joachims with the own created SVMLight algorithm [Joa98].
B LINEAR CLASSIFIERS

B.1 The Perceptron

The first iterative algorithm for learning linear classification is the procedure proposed by Frank Rosenblatt in 1956 for the Perceptron [Nel00].

In the neural network literature another view on the Perceptron is given, which is mostly more understandable (see figure B.1.1).

It starts with an initial weight vector $\mathbf{w}_0$ (usually all zero) and adapts it each time a training example is misclassified by the current weights.

A fact that needs to be stressed here is, that the weight vector and the bias are updated directly in the algorithm, something that is referred to as
the **primal** form in contrast to an alternative **dual** representation which will be introduced below.

The whole procedure used is guaranteed to converge if and only if the training points are able to be classified by a hyperplane. In this case the data is said to be **linearly separable**. If not so the weights (and the bias) are updated infinitely each time a point is misclassified and so the algorithm isn’t able to converge and only jumps from one instable state to the next. In this case the data is **nonseparable**.

For a detailed description of the algorithms see [Nel00].

---

Given a linearly separable training set \( S = \{(x_1, y_1), \ldots, (x_n, y_n)\} \) with \( X \subseteq \mathbb{R}^n \), \( Y = \{-1, 1\} \), the learning rate \( \eta \in \mathbb{R}^+ \) and the initial parameters \( w_0 = 0, b_0 = 0, k = 0 \)

\[
R = \max_{x_i} \|x_i\|
\]

**Repeat**

For \( i = 1 \) to \( n \)

- **If** \( y_i (\langle w_k \cdot x_i \rangle) + b_k \leq 0 \) **// mistake**
  - \( w_{k+1} \leftarrow w_k + \eta y_i x_i \)
  - \( b_{k+1} \leftarrow b_k + \eta y_i R^2 \)
  - \( k++ \)

**end if**

**end for**

**until** no mistakes in **for** loop

**Return** \( k, (w_k, b_k), k \) is the number of mistakes

---

**Figure B.1.2:** The Perceptron Algorithm for training in **primal** form

The training of figure B.1.2 leads to the following decision function for some unseen data \( z \), that needs to be classified:

\[
h(z) = \text{sgn}(\langle w_k \cdot z \rangle + b_k) = \text{sgn}(\sum w_i z_i)
\]

One can see in this algorithm that the perceptron ‘simply’ works by adding misclassified positive \((y = 1)\) training examples or subtracting misclassified negative \((y = -1)\) ones to an initial weight vector \( w_0 \).

So, if we assume an initial weight vector as the zero vector, overall the resulting weight vector is a **linear combination** of all training points:
\[ w = \sum_{i=1}^{n} \alpha_i y_i x_i \]  
\[ (B.1.1) \]

with all \( \alpha_i \geq 0 \), because the sign is already given by the corresponding \( y_i \).
The main property of all \( \alpha_i \) is, that their value is proportional to the number
of times a misclassification of \( x_i \) has caused the weight to be updated.

Therefore once the linearly separable training set \( S \) has been correctly
classified by the Perceptron and the weight vector has converged to its’
stable state one can think of the newly introduced vector \( \alpha \) as an alterna-
tive representation of the primal form, the so called dual form in dual co-
ordinates:

\[
f(x) = \langle w \cdot x \rangle + b = \sum w_i x_i + b \\
= \langle \sum \alpha_i y_i x_i \cdot x \rangle + b \\
= \sum \alpha_i y_i \langle x_i \cdot x \rangle + b
\]  
\[ (B.1.2) \]

And so the perceptron algorithm can be rewritten in the dual form as
shown in figure B.1.3.

<table>
<thead>
<tr>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>For ( i = 1 ) to ( n )</td>
</tr>
<tr>
<td>[ \text{If } y_i (\sum_{j=1}^{n} \alpha_j y_j \langle x_j \cdot x_i \rangle + b) \leq 0 \ // \text{mistake} ]</td>
</tr>
<tr>
<td>( \alpha_i \leftarrow \alpha_i + 1 )</td>
</tr>
<tr>
<td>( b \leftarrow b + y_i R^2 )</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>until no mistakes in for loop</td>
</tr>
<tr>
<td>Return ( (\alpha, b) ) for defining the decision function</td>
</tr>
</tbody>
</table>

**Figure B.1.3:** The Perceptron Algorithm for training in dual form
The learning rate is omitted here, because it only changes the scaling of the hyperplanes, but does not affect the algorithm with a starting vector of zero.
Overall the decision function in dual representation for unseen data \( z \) is given by:

\[
h(z) = \text{sgn}(\langle w \cdot z \rangle + b)
\]
\[
= \text{sgn}(\sum \alpha_i y_i \langle x_i \cdot z \rangle + b) \tag{B.1.3}
\]
\[
= \text{sgn}(\sum \alpha_i y_i \langle x_i \cdot z \rangle + b)
\]

This alternative representation of the primal Perceptron Algorithm and the corresponding decision function has many interesting and important properties. Firstly the points in the training set which were harder to learn have larger \( \alpha_i \), but the most important thing that needs to be stressed here is the fact, that the training points \( x_i \) (and so the unseen points) only enter the algorithm in form of the inner product \( \langle x_i \cdot x \rangle \), which will have an enormous impact on the discussed algorithm(s) used by the Support Vector Machines, there referenced to as a so called Kernel.
B.2 A calculated example with the Perceptron Algorithm

The sourcecode for this example in dual form, written in Matlab, can be obtained here (see also B.1):

Matlab Files\Perceptron\DualPerceptron.m

The already defined workspace variables are here:
Matlab Files\Perceptron\DualPerceptronVariables OR AND.mat

For a better understanding of linear separability, we have a look at the most common used binary functions: AND, OR and XOR.

The calling convention is: \([\text{weights bias alphas}] = \text{DualPerceptron}(X,Y)\).

For the OR- and the AND-datasets are both linearly separable while the XOR-data cannot be separated by means of one line. In these three cases the hyperplane is a line, because the inputspace is 2-dimensional (see Chapter 5).

Figure B.2.1: Examples for linearly separable and non separable data
Definition B.2.1 (Separability):

A training set \( S = \{(x_i, y_i) : x_i \in \mathbb{R}^n, y_i \in \{-1, +1\}\} \) is called separable by the hyperplane \( \langle w \cdot x \rangle + b = 0 \), if there exists both a vector \( w \) and a constant \( b \), such that following conditions are always true:

\[
\langle w \cdot x \rangle + b > 0 \quad \text{for} \quad y_i = 1 \\
\langle w \cdot x \rangle + b < 0 \quad \text{for} \quad y_i = -1
\]

The hyperplane defined by \( w \) and \( b \) is called a \textit{separating hyperplane}.

In detail we only calculate the OR case:

After the dual-perceptron-algorithm has converged to its stable state, the vector \( \alpha \) consist of \((7, 3, 3, 0)\)' and the bias has a value of -2.

So now we are able to define the weight vector (see equation B.1.1):

\[
w = 7 \cdot (-1) \cdot (0, 0)' + 3 \cdot 1 \cdot (0, 1)' + 3 \cdot 1 \cdot (1, 0)' + 0 \cdot 1 \cdot (1, 1)' = (3, 3)'
\]

The whole function of the hyperplane separating the OR-Dataset, here a line, is then defined as follows:

\[
f(x) = \langle w \cdot x \rangle + b = \sum w_i x_i + b = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \cdot x - 2 = 3x_1 + 3x_2 - 2
\]

If you test the decision function of B.1.3 with the values \( x \) of the just used OR-table in figure B.1.4, the classification of each point is correct.

E.g. Test of \( x_1 = (0, 0)' \) and \( x_3 = (1, 0)' \):

\[
\text{sgn}\left(\begin{pmatrix} 3 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 2\right) = \text{sgn}(3*0 + 3*0 - 2) = \text{sgn}(-2) = -1
\]

\[
\text{sgn}\left(\begin{pmatrix} 3 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} - 2\right) = \text{sgn}(3*1 + 3*0 - 2) = \text{sgn}(1) = 1
\]
C CALCULATION EXAMPLES

C.1 Chapter 4

- *Lagrangian method on a constrained function in two variables and a graphical way to find a solution*:

  We search the local extremes of the function

  \[ f(x, y) = x^2 + 2y^2 \]

  constrained by

  \[ g(x, y) = x + y = 3. \]

  As a first intuition we choose a graphical way to do this:

  First draw the constraint into the x-y-plane, then insert the isoquants (level lines) of the function \( f \) and last search level lines, which are cut by the constraint, to get an approximation where the optimum is.

  Isoquants or level lines are defined as seen in figure C.1.1.

![Figure C.1.1: The function \( f(x, y) = e^{-x^2} \cdot e^{-y^2} \) and the corresponding level lines](image)
The above technique is shown in figure C.1.2.

And now the solution with the Lagrangian method. As seen in chapter 4, the Lagrangian for a objective function $f(x, y)$ in two variables with one constraint $g(x, y) = c$ is defined as:

$$L(x, y, \alpha) = f(x, y) + \alpha(c - g(x, y))$$

The necessary conditions for a optimal solution can then be stated as (find stationary point(s)):

$$\frac{\partial}{\partial x} L(x, y, \alpha) = L_x = f_x + \alpha g_x = 0$$
$$\frac{\partial}{\partial y} L(x, y, \alpha) = L_y = f_y + \alpha g_y = 0$$
$$\frac{\partial}{\partial \alpha} L(x, y, \alpha) = L_\alpha = c - g(x, y) = 0$$

Therefore the example can be reformulated in that way:

$$L(x, y, \alpha) = (x^2 + 2y^2) + \alpha(3 - x - y)$$

And to find the stationary point(s):
This (linear) system of equalities has following solution:

\[ x = 2, \ y = 1 \text{ und } \alpha = 4. \]

So the only stationary point of \( f(x, y) \) constrained by \( g(x, y) \) is \( x_0 = (2; 1) \).

- Lagrangian method on a constrained function in three variables and two constraints.

We search the stationary points of the function

\[ f(x, y, z) = (x - 1)^2 + (y - 2)^2 + 2z^2 \]

constrained by

\[ x + 2y = 2 \text{ and } y - x = 3 \]

Recall the generalized Lagrangian function for equality constraints in chapter 3:

\[ L(x_1, x_n; \alpha_1, \ldots, \alpha_k) = f(x_1, x_n) + \sum_{i=1}^{k} \alpha_i (c_i - g_i(x_1, x_n)) \]

for a function \( f \) of \( n \) variables and \( k \) equality constraints \( g_i \) of the form \( g_i(x_1, \ldots, x_n) = c_i \).

So the Lagrangian function for the example is:

\[ L(x, y, z, \alpha, \beta) = ((x - 1)^2 + (y - 2)^2 + 2z^2) + \alpha(2 - x - 2y) + \beta(3 - y + z) \]

And the conditions for stationary points of \( L \) can be stated as:

\[
\begin{align*}
L_x &= 2(x - 1) - \alpha = 0 \\
L_y &= 2(y - 2) - 2\alpha - \beta = 0 \\
L_z &= 4z + \beta = 0 \\
L_\alpha &= 2 - x - 2y = 0 \\
L_\beta &= 3 - y + z = 0
\end{align*}
\]
And again we get a (linear) system with 5 unknowns in 5 variables, which can be easily solved and get as the only solution:

\[
x = -\frac{6}{7}; \quad y = \frac{10}{7}; \quad z = -\frac{11}{7}; \quad \alpha = -\frac{26}{7}; \quad \beta = \frac{44}{7}
\]

And so the only stationary point of \( f(x, y, z) \) with above constraints is \( x_0 = \left(-\frac{6}{7}, \frac{10}{7}, -\frac{11}{7}\right) \).

### C.2 Chapter 5

- **Equation 5.1:** \[ f(x) = \langle \mathbf{w} \cdot \mathbf{x} \rangle + b = \sum w_i x_i + b \]

  With \( \mathbf{w} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \ b = -3 \) and \( \mathbf{x} = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \):

  \[
  \Rightarrow \quad f(x) = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 5 \end{pmatrix} - 3 = 1 \cdot 2 + 3 \cdot 5 - 3 = 15
  \]

- **Definition 5.1 (Margin):** Normalisation of \( \mathbf{w} \) and \( b \) by \( \mathbf{w} = \frac{1}{\|\mathbf{w}\|} \mathbf{w} \) and \( b = \frac{1}{\|\mathbf{w}\|} b \).

  With \( \mathbf{w} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}, \ b = -3 \):

  \[
  \|\mathbf{w}\| = \sqrt{2^2 + 5^2} = \sqrt{29} \quad \Rightarrow \quad \mathbf{w}_{\text{norm}} = \frac{1}{\sqrt{29}} \begin{pmatrix} 2 \\ 5 \end{pmatrix}, \ b_{\text{norm}} = \frac{-3}{\sqrt{29}}
  \]

  So \( \|\mathbf{w}_{\text{norm}}\| = \|b_{\text{norm}}\| = 1. \)
In words normalising means scaling a vector to a length of 1, e.g. \[
\begin{bmatrix} 1 \\ 1 \end{bmatrix}
\]
can be seen as the diagonal in the unit quadrangle and therefore has a length of \(\sqrt{2}\), which is the same as \(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\). So scaling by \(\frac{1}{\text{length}}\) performs the step.
D SMO PSEUDO CODES

D.1 Pseudo Code of original SMO

target = desired output vector
point = training point matrix

procedure takeStep(i1,i2)
  if (i1 == i2) return 0

  alph1 = Lagrange multiplier for i1
  y1 = target[i1]
  E1 = SVM output on point[i1] - y1 (check in error cache)
  m = y1*y2

  Compute L, H

  if (L == H)
    return 0

  k11 = kernel(point[i1],point[i1])
  k12 = kernel(point[i1],point[i2])
  k22 = kernel(point[i2],point[i2])

  eta = 2*k12-k11-k22

  if (eta < 0)
    {a2 = alph2 - y2*(E1-E2)/eta
     if (a2 < L) a2 = L
     else if (a2 > H) a2 = H
    }

  else
    {
      Lobj = objective function at a2=L
      Hobj = objective function at a2=H

      if (Lobj > Hobj+eps)
        a2 = L
      else if (Lobj < Hobj-eps)
        a2 = H
      else
        a2 = alph2
    }

  if (a2 < 1e-8)
    a2 = 0
else if (a2 > C·1e-8)
    a2 = C
  if (|a2-alph2| < eps*(a2+alph2+eps))
      return 0

a1 = alph1+m*(alph2-a2)

Update threshold to reflect change in Lagrange multipliers
Update weight vector to reflect change in a1 & a2, if linear SVM
 Update error cache using new Lagrange multipliers
Store a1 in the alpha array
Store a2 in the alpha array

return 1
endprocedure

procedure examineExample(i2)
  y2 = target[i2]
  alph2 = Lagrange multiplier for i2
  E2 = SVM output on point[i2] - y2 (check in error cache)
  r2 = E2*y2

  if ((r2 < -tol && alph2 < C) || (r2 > tol && alph2 > 0))
    if (number of non-zero & non-C alpha > 1)
      i1 = result of second choice heuristic
      if takeStep(i1,i2)
          return 1
      }
  }

  loop over all non-zero and non-C alpha, starting at random point
  { 
    i1 = identity of current alpha
    if takeStep(i1,i2)
      return 1
    }

  loop over all possible i1, starting at a random point
  { 
    i1 = loop variable
    if takeStep(i1,i2)
      return 1
    }

  return 0
endprocedure
main routine:

initialize alpha array to all zero
initialize threshold to zero
numChanged = 0;
examineAll = 1;

while (numChanged > 0 | examineAll)
{
    numChanged = 0;

    if (examineAll)
        loop I over all training examples
        numChanged += examineExample(I)
    else
        loop I over examples where alpha is not 0 & not C
        numChanged += examineExample(I)

    if (examineAll == 1)
        examineAll = 0
    else if (numChanged == 0)
        examineAll = 1
}

D.2  Pseudo Code of Keerthi’s improved SMO

target = desired output vector
point = training point matrix
fcache = cache vector for Fi values

% Note: The definition of Fi is different from the Ei in Platt’s SMO algorithm.
% The Fi does not subtract any threshold.

procedure takeStep(i1, i2)
% Much of this procedure is same as in Platt’s original SMO pseudo code

    if (i1 == i2) return 0

    alph1 = Lagrange multiplier for i1
    y1 = target[i1]
    F1 = fcache[i1]
    m = y1*y2
    Compute L, H

    if (L == H) return 0
K11 = kernel(point[i1], point[i1])
K12 = kernel(point[i1], point[i2])
K22 = kernel (point[i2], point[i2])
eta = 2*K12-K11-K22

if (eta < 0)
{
    a2 = alph2 - y2*(F1-F2)/eta
    if (a2 < L) a2 = L
    else if (a2 > H) a2 = H
}
else
{
    Lobj  = objective function at a2=L
    Hobj = objective function at a2=H
    if (Lobj > Hobj+eps)
        a2 = L
    else if (Lobj < Hobj - eps)
        a2 = H
    else
        a2 = alph2
}

if ( |a2-alph2| < eps*(a2+alph2+eps) )
    return 0

a1 = alph1+m*(alph2-a2)

Update weight vector to reflect change in a1 & a2, if linear SVM
Update fcache[i] for i in I_0 using new Lagrange multipliers
Store a1 and a2 in the alpha array

% The update below is simply achieved by keeping and updating infor-
% mation about alpha_i being 0, C or in between them. Using this to-
% gether with target[i] gives information as to which index set I belongs
Update I_0, I_1, I_2, I_3 and I_4

% Compute updated F values for i1 and i2 ...

fcache[i1] = F1 + y1*(a1-alph1)*k11 + y2*(a2-alph2)*k12
fcache[i2] = F2 + y1*(a1-alph1)*k12 + y2(a2-alph2)*k22

Compute (i_low, b_low) and (i_up, b_up) by applying equations (A) and (B)
using only i1, i2 and indices in I_0

return 1

endprocedure
procedure examineExample(i2)

    y2 = target[i2]
    alph 2 = Lagrange multiplier for i2

    if (i2 is in I_0)
        { 
            F2 = fcache[i2]
        }

    else
        { 
            compute F2 = F_i2 and set fcache[i2] = F2

            % Update (b_low, i_low) or (b_up, i_up) using (F2, i2) ...

            if (((i2 is in I_1 or I_2) && (F2 < b_up))
                b_up = F2,  i_up = i2
            else if (((i2 is in I_3 or I_4) && (F2 > b_low))
                b_low = F2,  i_low = i2
            }

            % Check optimality using current b_low and b_up and, if violated, find an
            % index i1 to do joint optimization with i2 ....

            optimality = 1

            if (i2 is in I_0, I_1 or I_2)
                { 
                    if (b_low – F2 > 2*tol)
                        optimality = 0,  i1 = i_low
                }

            if (i2 is in I_0, I_3 or I_4)
                { 
                    if (F2 – b_up > 2*tol)
                        optimality = 0,  i1 = i_up
                }

            if (optimality == 1)
                return 0

            % For i2 in I_0 choose the better i1 ...

            if (i2 is in I_0)
                { 
                    if (b_low – F2 > F2 – b_up)
                        i1 = i_low
                    else
                        i1 = i_up
                }

            if takeStep(i1, i2)
                return 1
            else
main routine for Modification 1 (same as SMO):

initialize alpha array to all zero
initialize $b_{up} = -1$, $i_{up}$ to any index of class 1
initialize $b_{low} = 1$, $i_{low}$ to any index of class 2
set $f_{cache}[i_{low}] = 1$ and $f_{cache}[i_{up}] = -1$
numChanged = 0;
examineAll = 1;

while (numChanged > 0 | examineAll)
{
    numChanged = 0;

    if (examineAll)
    {
        loop $I$ over all training examples
        numChanged += examineExample($I$)
    }
    else
    {
        loop $I$ over $I_0$
        numChanged += examineExample($I$)
    }

    % It is easy to check if optimality on $I_0$ is attained …
    if ($b_{up} > b_{low} - 2*tol$) at any $I$
        exit the loop after setting numChanged = 0
    }

    if (examineAll == 1)
        examineAll = 0
    else if (numChanged == 0)
        examineAll = 1
    }

main routine for Modification 2:

initialize alpha array to all zero
initialize $b_{up} = -1$, $i_{up}$ to any index of class 1
initialize $b_{low} = 1$, $i_{low}$ to any index of class 2
set $f_{cache}[i_{low}] = 1$ and $f_{cache}[i_{up}] = -1$
numChanged = 0;
examineAll = 1;

while (numChanged > 0 | examineAll)
{
    numChanged = 0;

if (examineAll)
{
    loop I over all training examples
    numChanged += examineExample(I)
}
else
%
The following loop is the only difference between the two SMO modific-
% fications. Whereas, in modification 1, the inner loop selects i2 from I_0
%
sequentially, here i2 is always set to the current i_low and i1 is set to
%
the current i_up; clearly, this corresponds to choosing the worst vio-
%
lating pair using members of I_0 and some other indices.
{
    inner_loop_success = 1;
    do until ((b_up > b_low-2*tol) | inner_loop_success == 0)
    {
        i2 = i_low
        y2 = target(i2)
        alph2 = Lagrange multiplier for i2
        F2 = fcache[i2]
        Inner_loop_success = takeStep(i_up, i_low)
        numChanged += inner_loop_success
    }
    numChanged = 0
}