Averaging of Support Vector Machines for Large Scale Learning
Giving the constants a beating

Jochen Garcke
Matheon and Technische Universität Berlin
decision function of SVM is of form

\[ y := \text{sign}(f(x)) \]

with

\[ f(x) = \sum_{i=1}^{N} \alpha_i y_i k(x_i, x) + b, \]

- \( y_i \in \{-1, 1\} \) class labels
- \( x_i \) positions of training data
- \( N \) is the number of training data
- \( \alpha_i \) are the coefficients of the solution \( f(x) \) in regard to each kernel function \( k(x_i, x) \), where \( k(\cdot, \cdot) \) is a positive definite function
- data points with non-zero \( \alpha_i \) are support vectors (SV)
- hopefully many \( \alpha_i \) are zero, small number of SV
- number of SVs grows as a linear function of \( N \) [Steinwart:2003]
computing coefficients in standard way involves solution of dual optimization problem

\[
\min_{\alpha_i} \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{N} \alpha_i
\]

subject to

\[
\sum_{i=1}^{N} \alpha_i y_i = 0 \text{ and } 0 \leq \alpha_i \leq C \quad \forall i,
\]

with some parameter \( C \)

- essential difference between linear and non-linear kernel case
- for non-linear kernels scaling looks like quadratic or even worse
- complexity constants get bad when kernel matrix cannot be cached
Averaging of SVMs for Large Scale Data

- assume quadratic scaling in number of data
- first view
  - we now separate into $p$ data sets of size $N/p$
  - compute $p$ partial SVMs solutions
    \[ f_j^i(x) = \sum_{i=1}^{N/p} \alpha_j^i y_j^i k(x_j^i, x) + b_j \]
    - we now have $\mathcal{O}(p \cdot N^2/p^2)$ complexity instead of $\mathcal{O}(N^2)$
    - gain a constant $p$ in complexity but is of same order in regard to $N$
- second view
  - we now separate into $p = N/k$ data sets of size $k$
  - compute $N/k$ partial SVMs solutions
    \[ f_j^i(x) = \sum_{i=1}^{k} \alpha_j^i y_j^i k(x_j^i, x) + b_j \]
    - we now have $\mathcal{O}(N/k \cdot k^2) = \mathcal{O}(N \cdot k)$ complexity instead of $\mathcal{O}(N^2)$
    - complexity gets linear in $N$
Some formal observations

- decision function is now just

\[
\text{sign} \left( \frac{1}{p} \sum_{j=1}^{p} f^j(x) \right)
\]

- one could write overall solution as (with \( b = \sum_{j=1}^{p} b^j \))

\[
f(x) = \sum_{j=1}^{p} \sum_{i=1}^{N/p} \frac{\alpha^j_i y^i_j}{p} k(x^i_j, x) + b,
\]

- regard it as an approximation to full optimisation problem?
- not really!
  - partial SVMs have no reason to be good on training data of others SVMs
  - performance will be like on test data (maybe avoids overfitting?)
  - concept of regularisation by approximation (relation to early stopping)
Other computational advantages

- use smaller problems which can keep kernel matrix in memory: smaller constants in overall complexity
- easy to implement: loop over partial SVMs
- embarrassingly easy to parallelise in a distributed memory setup
- to solve partial SVMs use favourite solver/package
- we employ libsvm and the shogun package in our experiments
- employ Gaussian kernel $\exp\left(-\frac{\|x - y\|^2}{\tau}\right)$
- width $\tau$ and regularisation $C$ via testing on part of training data
Results in the challenge, alpha

Av SVM - Give the constants a beating
Results in the challenge, alpha

Av SVM - Give the constants a beating
Results in the challenge, beta, gamma

Av SVM - Give the constants a beating
Some comparisons on the alpha data set

DISCLAIMER

Results on one data set, might not be significant, but interesting

- take last 50,000 of training data as test data
- train on first 50,000 and 100,000
- \( C = 2^{14}, \tau = 2^{16} \)
- \( \|x - y\|^2 \) evaluation in single, rest in double

<table>
<thead>
<tr>
<th></th>
<th>train (sec)</th>
<th>test (sec)</th>
<th>FalseRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av 10x5000</td>
<td>384</td>
<td>1145</td>
<td>18.55%</td>
</tr>
<tr>
<td>Full SVM 50.000, ( \tau = 2^{16} )</td>
<td>16378</td>
<td>1009</td>
<td>18.07%</td>
</tr>
<tr>
<td>Full SVM 50.000, ( \tau = 2^{18} )</td>
<td>17485</td>
<td>933</td>
<td>16.84%</td>
</tr>
<tr>
<td>Av 10x10000</td>
<td>1871</td>
<td>2285</td>
<td>16.92%</td>
</tr>
<tr>
<td>Av 20x10000</td>
<td>3828</td>
<td>4727</td>
<td>16.49%</td>
</tr>
<tr>
<td>Av 10x20000</td>
<td>9242</td>
<td>4460</td>
<td>15.25%</td>
</tr>
</tbody>
</table>

Av SVM - Give the constants a beating
Do we really make use of all subsets?

<table>
<thead>
<tr>
<th>i-th part</th>
<th>10x5000</th>
<th>10x10000</th>
<th>20x10000</th>
<th>10x20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.26%</td>
<td>22.64%</td>
<td>19.89%</td>
<td>20.87%</td>
</tr>
<tr>
<td>2</td>
<td>21.93%</td>
<td>19.91%</td>
<td>18.25%</td>
<td>18.36%</td>
</tr>
<tr>
<td>3</td>
<td>20.60%</td>
<td>18.82%</td>
<td>17.45%</td>
<td>17.11%</td>
</tr>
<tr>
<td>4</td>
<td>19.99%</td>
<td>18.25%</td>
<td>17.16%</td>
<td>16.57%</td>
</tr>
<tr>
<td>5</td>
<td>19.45%</td>
<td>17.81%</td>
<td>16.92%</td>
<td>16.02%</td>
</tr>
<tr>
<td>6</td>
<td>19.20%</td>
<td>17.45%</td>
<td>16.74%</td>
<td>15.80%</td>
</tr>
<tr>
<td>7</td>
<td>18.94%</td>
<td>17.33%</td>
<td>16.65%</td>
<td>15.55%</td>
</tr>
<tr>
<td>8</td>
<td>18.88%</td>
<td>17.16%</td>
<td>16.59%</td>
<td>15.41%</td>
</tr>
<tr>
<td>9</td>
<td>18.72%</td>
<td>16.98%</td>
<td>16.56%</td>
<td>15.26%</td>
</tr>
<tr>
<td>10</td>
<td>18.55%</td>
<td>16.92%</td>
<td>16.48%</td>
<td>15.25%</td>
</tr>
</tbody>
</table>
Efficient kernel evaluation

- Gaussian kernel $\exp(-\|x - y\|^2/\tau)$
- with large number of attributes time dominated by $\|x - y\|$
- $\exp$ and divide should be done efficient in the math lib
- shogun uses double precision per default
- Gaussian kernel evaluation:
  
  ```
  double result = 0;
  for (INT i = 0; i < alen; i++)
      result += CMath::sq(avec[i] - bvec[i]);
  result = exp(-result / width);
  ```
- precision of input data is float, no need for double there
- double kernel: training 547 / test 2533
- float kernel: training 451 / test 1802
Can we achieve more improvement?

- modern CPUs support SSE2 (Streaming SIMD Extensions 2)
- exploit this

```c
static inline float l2_diff_float(int n, const float*x, const float*y) {
    [...]
    for (; n>3; n-=4) {
        const __m128 a = _mm_loadu_ps(x);
        const __m128 b = _mm_loadu_ps(y);
        const __m128 a_minus_b = _mm_sub_ps(a, b);
        const __m128 a_minus_b_sq = _mm_mul_ps(a_minus_b, a_minus_b);
        l2_diff = _mm_add_ps(l2_diff, a_minus_b_sq);
        x+=4;
        y+=4;
    }
    [...]
```
Can we do more?

- use different compiler
- test pathscale compiler suite for Opteron on fortran code

```fortran
real*4 function norm_single(n, x, y)
integer n
real*4 x(*), y(*)

integer i

do i = 1, n
    norm_single = norm_single + (x(i) - y(i))**2
endo

end function norm_single
```

- link the .o to shogun
- assembler code uses SSE2 and heuristics with some probability
What do we gain?

<table>
<thead>
<tr>
<th>code</th>
<th>train</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>double, gcc</td>
<td>547</td>
<td>2533</td>
</tr>
<tr>
<td>double, pathscale</td>
<td>526</td>
<td>2250</td>
</tr>
<tr>
<td>float, gcc</td>
<td>451</td>
<td>1802</td>
</tr>
<tr>
<td>float, sse2 by hand</td>
<td>418</td>
<td>1480</td>
</tr>
<tr>
<td>float, pathscale</td>
<td>384</td>
<td>1145</td>
</tr>
</tbody>
</table>

- code of pathscale compiler is non-deterministic
- two runs: false rate 18.55% to 18.56% objective value 346725295 to 346725351
- no large difference, but ...
- effect not observed for double precision
Further possible improvements

- compile shogun with pathscale: didn’t work on first try
- still need to check: memory not aligned to a 16-byte boundary will incur significant penalty
- batch evaluation of more test data might use blas routines for matrix computations due to

$$\|x - y\|^2 = x \cdot x + y \cdot y - 2x \cdot y$$

- use GPUs (see talk by Catanzaro, Sundaram, and Keutzer yesterday)
Conclusions

- simple averaging of partial SVMs for nonlinear kernels
- gives good baseline results
- while using more data it can be better as full SVM in less time

- kernel evaluation is the core bottleneck in optimisation
- different coding/compilation of kernel function can gain almost factor 2 in comparison to standard implementation
- if it comes to runtime have a thought about kernel computation

- have a try with different compiler (intel, pathscale, pgi) than gcc
- use vendor supplied blas/lapack if possible (Intel MKL, AMD ACML)