Large Scale Learning with String Kernels

Sören Sonnenburg
Fraunhofer FIRST.IDA, Berlin

joint work with
Gunnar Rätsch, Konrad Rieck

Fraunhofer Institut Rechnerarchitektur und Softwaretechnik

Outline

Large Scale Problems

- Text Classification (Spam, Web-Spam, Categorization)
  - Task: Given N documents, with class label ± 1, predict text type.
- Security (Network Traffic, Viruses, Trojans)
  - Task: Given N executables, with class label ± 1, predict whether executable is a virus.
- Biology (Promoter, Splice Site Prediction)
  - Task: Given N sequences around Promoter/Splice Site (label $+1$) and fake examples (label -1), predict whether there is a Promoter/Splice Site in the middle

$\Rightarrow$ **Approach: String kernel + Support Vector Machine**

$\Rightarrow$ **Large N is needed to achieve high accuracy (i.e. $N = 10^7$)**

Formally

- Given:
  - N training examples $(\mathbf{x}_i, y_i) \in (\mathcal{X}, \pm 1)$, $i = 1 \dots N$
  - string kernel $K(\mathbf{x}, \mathbf{x}') = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{x}')$
- Examples:
  - words-in-a-bag-kernel
  - k-mer based kernels (Spectrum, Weighted Degree)
- Task:
  - Train Kernelmachine on Large Scale Datasets, e.g. $N = 10^7$
  - Apply Kernelmachine on Large Scale Datasets, e.g. $N = 10^9$

String Kernels

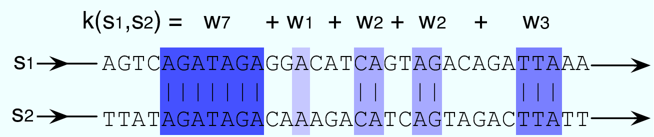
- Spectrum Kernel (with mismatches, gaps)

$$K(\mathbf{x},\mathbf{x}') = \Phi_{sp}(\mathbf{x}) \cdot \Phi_{sp}(\mathbf{x}')$$

x AAACAAATAAGTAACTAATCTTTTAGGAAGAACGTTTCAACCATTTTGAG

x' TACCTAATTATGAAATTAAATTTCAGTGTGCTGATGGAAACGGAGAAGTC

- Weighted Degree Kernel (with shift)



For string kernels $\mathcal{X}$ discrete space and $\Phi(x)$ sparse

Kernel Machine

Kernel Machine Classifier:

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}) + b\right)$$

To compute output on all M examples:

$$\forall j = 1, \ldots, M: \quad \sum_{i=1}^{N} \alpha_i y_i \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) + b$$

Computational effort:

- Single $\mathcal{O}(NT)$ (T time to compute the kernel)
- All $\mathcal{O}(NMT)$

$\Rightarrow$ **Costly!**

$\Rightarrow$ **Used in training and testing - worth tuning.**

$\Rightarrow$ **How to further speed up if $T = dim(\mathcal{X})$ already linear?**

Outline

Linadd Speedup Idea

Key Idea: Store $\mathbf{w}$ and compute $\mathbf{w} \cdot \Phi(\mathbf{x})$ *efficiently*

$$\sum_{i=1}^{N} \alpha_i y_i \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = \underbrace{\sum_{i=1}^{N} \alpha_i y_i \Phi(\mathbf{x}_i)}_{\mathbf{w}} \cdot \Phi(\mathbf{x}_j) = \mathbf{w} \cdot \Phi(\mathbf{x}_j)$$

When is that possible ?

1. $\mathbf{w}$ has low dimensionality and sparse (e.g. 4^8 for Feature map of Spectrum Kernel of order 8 DNA)
2. $\mathbf{w}$ is extremely sparse although high dimensional (e.g. 10^{14} for Weighted Degree Kernel of order 20 on DNA sequences of length 100)

Effort: $\mathcal{O}(MT')$ $\Rightarrow$ **Potential speedup of factor** N

Technical Remark

Treating w

- $\mathbf{w}$ must be accessible by some index u (i.e. $u = 1 \dots 4^8$ for 8-mers of Spectrum Kernel on DNA or word index for word-in-a-bag kernel)
- Needed Operations
  - `Clear:` $\mathbf{w} = \mathbf{0}$
  - `Add:` $w_u \leftarrow w_u + v$ (only needed $|W|$ times per iteration)
  - `Lookup:` obtain w_u (must be highly efficient)
- Storage
  - **Explicit Map** (store dense $\mathbf{w}$); `Lookup` in $\mathcal{O}(1)$
  - **Sorted Array** (word-in-bag-kernel: all words sorted with value attached); `Lookup` in $\mathcal{O}(\log(\sum_u I(w_u \neq 0)))$
  - **Suffix Tries, Trees**; `Lookup` in $\mathcal{O}(K)$

Datastructures - Summary of Computational Costs

Comparison of worst-case run-times for operations

- `clear` of $\mathbf{w}$
- `add` of all k-mers $\boldsymbol{u}$ from string $\mathbf{x}$ to $\mathbf{w}$
- `lookup` of all k-mers $\boldsymbol{u}$ from $\mathbf{x}'$ in $\mathbf{w}$

	Explicit map	Sorted arrays	Tries	Suffix trees
`clear`	$\mathcal{O}(\vert\Sigma\vert^d)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`add`	$\mathcal{O}(l_{\mathbf{x}})$	$\mathcal{O}(l_{\mathbf{x}} \log l_{\mathbf{x}})$	$\mathcal{O}(l_{\mathbf{x}} d)$	$\mathcal{O}(l_{\mathbf{x}})$
`lookup`	$\mathcal{O}(l_{\mathbf{x}'})$	$\mathcal{O}(l_{\mathbf{x}} + l_{\mathbf{x}'})$	$\mathcal{O}(l_{\mathbf{x}'} d)$	$\mathcal{O}(l_{\mathbf{x}'})$

Conclusions

- Explicit map ideal for small $|\Sigma|$
- Sorted Arrays for larger alphabets
- Suffix Arrays for large alphabets and order (overhead!)

Support Vector Machine

`Linadd` directly applicable when applying the classifier.

$$f(\mathbf{x}) = \mathrm{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \, \mathrm{k}(\mathbf{x}_i, \mathbf{x}) + b\right)$$

Problems

- $\mathbf{w}$ may still be huge $\Rightarrow$ fix by not constructing whole $\mathbf{w}$ but only blocks and computing batches

What about training?

- general purpose QP-solvers, Chunking, SMO
- optimize kernel (i.e. find $O(L)$ formulation, where $L = dim(\mathcal{X})$)
- Kernel Caching infeasable
  (for $N = 10^6$ only 125 kernel rows fit in 1GiB memory)

$\Rightarrow$ **Use `linadd` again: Faster + needs no kernel caching**

Derivation I

Analyzing Chunking SVMs (GPDT, SVM^{light}:)

Training algorithm (chunking):

while optimality conditions are violated **do**
  select q variables for the working set.
  solve reduced problem on the working set.
end while

- At each iteration, the vector $\boldsymbol{f}$, $f_j = \sum_{i=1}^{N} \alpha_i y_i \, \mathrm{k}(x_i, x_j)$, $j = 1 \ldots N$ is needed for checking termination criteria and selecting new working set (based on $\boldsymbol{\alpha}$ and gradient w.r.t. $\boldsymbol{\alpha}$).
- Avoiding to recompute $\boldsymbol{f}$, most time is spend computing "linear updates" on $\boldsymbol{f}$ on the working set W

$$f_j \leftarrow f_j^{old} + \sum_{i \in W} (\alpha_i - \alpha_i^{old}) y_i \, \mathrm{k}(x_i, x_j)$$

Derivation II

Use `linadd` to compute updates.

Update rule: $f_j \leftarrow f_j^{old} + \sum_{i \in W} (\alpha_i - \alpha_i^{old}) y_i \, \mathrm{k}(x_i, x_j)$
Exploiting $\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$ and $\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \Phi(\mathbf{x}_i)$:

$$f_j \leftarrow f_j^{old} + \sum_{i \in W} (\alpha_i - \alpha_i^{old}) y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j) = f_j^{old} + \mathbf{w}^W \cdot \Phi(\mathbf{x}_j)$$

($\mathbf{w}^W$ normal on working set)

Observations

- $q := |W|$ is very small in practice $\Rightarrow$ can effort more complex $\mathbf{w}$ and `clear`,`add` operation
- `lookups` dominate computing time

Algorithm

Recall we need to compute updates on $\mathbf{f}$ (effort $c_1|W|LN$):

$$f_j \leftarrow f_j^{old} + \sum_{i \in W} (\alpha_i - \alpha_i^{old}) y_i \, \mathrm{k}(x_i, x_j) \text{ for all } j = 1 \ldots N$$

Modified SVM^{light} using "LinAdd" algorithm (effort $c_2 \ell LN$, ℓ Lookup cost)

$f_j = 0$, $\alpha_j = 0$ for $j = 1, \ldots, N$
for $t = 1, 2, \ldots$ **do**
Check optimality conditions and stop if optimal, select working set W based on $\mathbf{f}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
solve reduced problem W and update $\boldsymbol{\alpha}$
clear $\mathbf{w}$
$\mathbf{w} \leftarrow \mathbf{w} + (\alpha_i - \alpha_i^{old}) y_i \Phi(\mathbf{x}_i)$ for all $i \in W$
update $f_j = f_j + \mathbf{w} \cdot \Phi(\mathbf{x}_j)$ for all $j = 1, \ldots, N$
end for

Speedup of factor $\frac{c_1}{c_2 \ell} |W|$

Parallelization

$f_j = 0$, $\alpha_j = 0$ for $j = 1, \dots, N$
for $t = 1, 2, \dots$ **do**
  Check optimality conditions and stop if optimal, select working set W based on $\mathbf{f}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
  solve reduced problem W and update $\boldsymbol{\alpha}$

  clear $\mathbf{w}$
  $\mathbf{w} \leftarrow \mathbf{w} + (\alpha_i - \alpha_i^{old}) y_i \Phi(\mathbf{x}_i)$ for all $i \in W$
  update $f_j = f_j + \mathbf{w} \cdot \Phi(\mathbf{x}_j)$ for all $j = 1, \dots, N$
end for

Most time is still spent in update step $\Rightarrow$ Parallize!

- transfer $\boldsymbol{\alpha}$ (or $\mathbf{w}$ depending on the communication costs and size)
- update of $\boldsymbol{f}$ is divided into chunks
- each CPU computes a chunk of $\boldsymbol{f}_I$ for $I \subset \{1, \dots, N\}$

Outline

Datasets

- Web Spam
  - Negative data: Use Webb Spam corpus `http://spamarchive.org/gt/` (350,000 pages)
  - Positive data: Download 250,000 pages randomly from the web (e.g. `cnn.com`, `microsoft.com`, `slashdot.org` and `heise.de`)
  - Use spectrum kernel $k = 4$ using **sorted arrays** on 100,000 examples train and test (average string length 30Kb, 4 GB in total, 64bit variables $\Rightarrow$ 30GB)

Web Spam results

Classification Accuracy and Training Time

N	100	500	5,000	10,000	20,000	50,000	70,000	100,000
Spec	2	97	1977	6039	19063	94012	193327	-
LinSpec	3	255	4030	9128	11948	44706	83802	107661
Accuracy	89.59	92.12	96.36	97.03	97.46	97.83	97.98	98.18
auROC	94.37	97.82	99.11	99.32	99.43	99.59	99.61	99.64

Speed and classification accuracy comparison of the spectrum kernel without (*Spec*) and with `linadd` (*LinSpec*)

Datasets

- Splice Site Recognition
  - Negative Data: 14,868,555 DNA sequences of fixed length 141 base pairs
  - Positive Data: 159,771 Acceptor Splice Site Sequences
  - Use WD kernel $k = 20$ (using **Tries**) and spectrum kernel $k = 8$ (using **explicit maps**) on $10,000,000$ train and 5,028,326 examples

Linadd for WD kernel

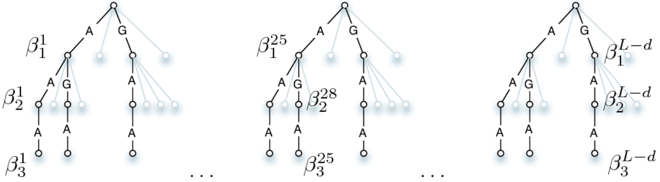
For linear combination of kernels:

$\sum_{j \in W}(\alpha_j - \alpha_j^{old}) y_j \, \mathrm{k}(x_i, x_j) \; (\mathcal{O}(Ld|W|N))$

AAACTAATTATGAAATTAAATTTCAGAGTGCTGATGGAAACGGAGAAGAA

- use one tree of depth d per position in sequence
- for `Lookup` use traverse one tree of depth d per position in sequence

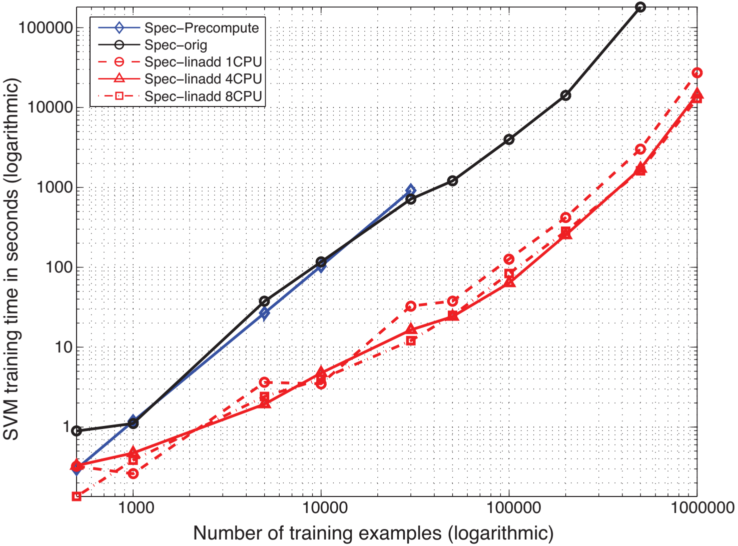
Example $d = 3$:



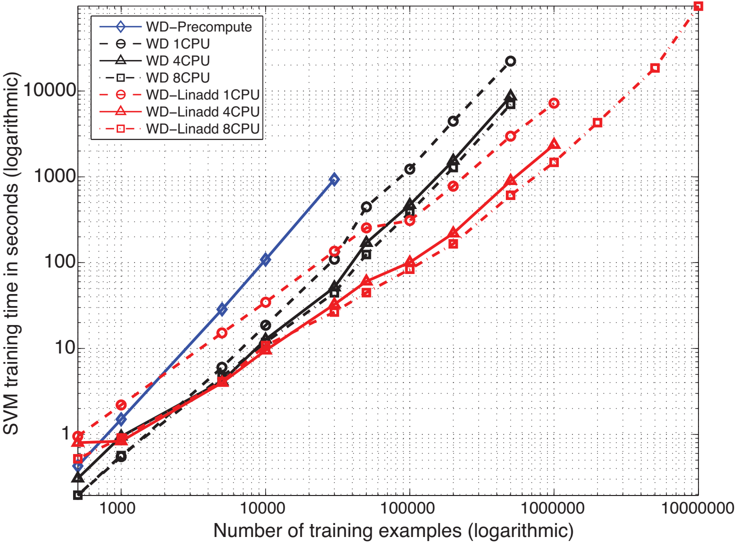
output for N sequences of length L in $\mathcal{O}(Ld \cdot N)$

(d depth of tree $\triangleq$ degree of WD kernel)

Spectrum Kernel on Splice Data



Weighted Degree Kernel on Splice Data



More data helps

N	*auROC*	*auPRC*
500	75.55	3.94
1,000	79.86	6.22
5,000	90.49	15.07
10,000	92.83	25.25
30,000	94.77	34.76
50,000	95.52	41.06
100,000	96.14	47.61

N	*auROC*	*auPRC*
200,000	96.57	53.04
500,000	96.93	59.09
1,000,000	97.19	63.51
2,000,000	97.36	67.04
5,000,000	97.54	70.47
10,000,000	97.67	72.46
10,000,000	96.03*	44.64*

Discussion

Conclusions

- General speedup trick (`clear`, `add`, `lookup` operations) for string kernels
- Shared memory parallelization, able to train on **10 million** human splice sites
- Linadd gives speedup of factor 64 (4) for Spectrum (Weighted Degree) kernel and 32 for MKL
- 4 CPUs further speedup of factor 3.2 and for 8 CPU factor 5.4
- parallelized 8 CPU linadd gives speedup of factor 125 (21) for Spectrum (Weighted Degree) kernel, up to 200 for MKL

Discussion

- State-of-the-art accuracy
- Could we do better by encoding invariances?

Implemented in SHOGUN `http://www.shogun-toolbox.org`

Machine Learning Open Source Software

To support the open source movement JMLR is proud to announce a new track on machine learning open source software.

Contributions to http://jmlr.org/mloss/ should be related to

- Implementations of machine learning algorithms,
- Toolboxes,
- Languages for scientific computing

and should include

- A 4 page description,
- The code,
- A recognised open source license.

Contribute to http://mloss.org the mloss repository!