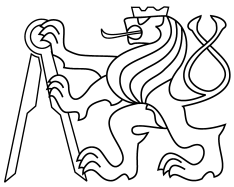
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY

RESEARCH REPORT

COFFIN : A Computational Framework for Linear SVMs

Sören Sonnenburg[1] and Vojtěch Franc[2]

soeren.sonnenburg@tu-berlin.de
xfrancv@cmp.felk.cvut.cz

[1]Berlin Institute of Technology, Franklinstr. 28/29, 10587 Berlin, Germany
[2]Czech Technical University in Prague, Technicka 2, 16627 Praha 6, Czech Republic

CTU–CMP–2009–23

December 24, 2009

Available at
ftp://cmp.felk.cvut.cz/pub/cmp/articles/franc/Sonnenburg-TR-2009-23.pdf

This research was supported by a Marie Curie European Reintegration Grant SEMISOL (PERG04-GA-2008-239455) within the 7th European Community Framework Programme.

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Abstract

In a variety of applications, kernel machines such as Support Vector Machines (SVMs) have been used with great success often delivering state-of-the-art results. Using the kernel trick, they work on several domains and even enable heterogeneous data fusion by concatenating feature spaces or multiple kernel learning. Unfortunately, they are not suited for truly large-scale applications since they suffer from the curse of supporting vectors, e.g., the speed of applying SVMs decays linearly with the number of support vectors. In this paper we develop COFFIN — a new training strategy for linear SVMs that effectively allows the use of on demand computed kernel feature spaces and virtual examples in the primal. With *linear* training and prediction effort this framework leverages SVM applications to truly large-scale problems: As an example, we train SVMs for human splice site recognition involving 50 million examples and sophisticated string kernels. Additionally, we learn an SVM based gender detector on 5 million examples on low-tech hardware and achieve beyond the state-of-the-art accuracies on both tasks.

1 Introduction

Many applications in e.g. Bioinformatics, IT-Security and Text-Classification come with *huge* amounts (e.g. millions or billions) of training examples, which are indeed *needed* to obtain state-of-the-art results. At the same time predictions need to be made on billions of data points demanding for linear time algorithms that additionally can be effectively parallelized. Thus computationally highly effective methods are needed that can cope with ever growing data sizes. While classical kernel machines

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i K(\mathbf{x}, \mathbf{x}_i) + b\right) \tag{1}$$

often deliver state-of-the-art results, they are not suited for truly large scale applications since they suffer from the curse of supporting vectors, i.e. the number of non-zero coefficients α_i above. The total evaluation complexity to predict t elements, for m_{sv} support vectors and kernel complexity c is $\mathcal{O}(tm_{sv}c)$. Since $m_{sv} = \mathcal{O}(m)$, i.e., is linear in the training set size Steinwart [2004], kernel machines are in big-O notation (and for many practical applications) not at all faster than k-nearest neighbor: The number of training examples/support vectors m_{sv} becomes dominant, especially if kernel computations are already linear. Reduced set methods [Schölkopf and Smola, 2002] partially alleviate this problem by enforcing a low number of non-zero α_i in a post-processing step. Nevertheless, the computational complexity of determining a reduced support vector set and the potential performance degradation and the still prevailing prediction complexity render them infeasible for truly large-scale learning applications. Collobert et al. [2006] show that using non-convex loss function can largely reduce the number of support vectors, however, this is paid with more tricky optimization of a non-convex objective function. Keerthi et al. [2006] propose a greedy algorithm which simultaneously selects the set of support vectors and optimizes over the parameters α_i in (1). Again, this method optimizes non-convex cost and it is applicable to problems of moderate size only.

It is also operation (1) that slows down training in prominent SVM packages and potentially has caused a shift in interest from kernel SVMs back to linear SVMs for large-scale applications. Many of the recently proposed linear SVM solvers, are very efficient and guaranteed to converge to

a ε-precise solution in $\mathcal{O}(m)$ (e.g., Joachims [2006]). In this paper we develop a training strategy for linear SVMs that effectively allows the use of on-demand computed kernel like non-linearity and of virtual examples in the primal and thus leverages SVM applications to truly large-scale problems: As an example, we train a linear SVM on a gender classification dataset of almost 5 million images on a plain notebook with just 4GB of memory and on a bioinformatics splice site recognition task of 50 million examples using a 185 million dimensional string kernel feature space approximation to the traditionally spanned feature space of size $n > 10^{14}$. The paper is structured as follows: In Section 2 we introduce the COmputational Framework For lINear svms (COFFIN). In Section 3 we evaluate our proposed approach COFFIN on two real-world data sets. Section 4 concludes the paper.

2 Leveraging Linear SVMs

Given labeled training examples $(\mathbf{x}_i, y_i)_{i=1}^m \in (\mathbb{R}^n \times \{-1, +1\})^m$ and a regularization constant $C > 0$, SVMs learn a linear classification rule $f(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$ by minimizing the following quadratic optimization problem

$$\min_{\mathbf{w},b} F(\mathbf{w}) := \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{i=1}^{m} \max\{0, 1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)\}.$$

Among the most prominent linear SVM solvers minimizing $F(\mathbf{w})$ are quasi-Newton methods (implemented in subBFGS; Yu et al. [2008]), (stochastic) subgradient descent algorithms (implemented in SGD and Pegasos; Bottou and Bousquet [2008], Shwartz et al. [2007]), dual coordinate descent (implemented in Liblinear; Fan et al. [2008]) and cutting plane based algorithms (implemented in SVMPerf, BMRM and OCAS; Joachims [2006], Teo et al. [2007], Franc and Sonnenburg [2009]). For the latter four, convergence guarantees exist and they have been proven to have linear training time, i.e., achieve a ε precise solution in $\mathcal{O}(m)$.

Underneath, these algorithms are applied by supplying a set of sparse vectors $\mathbf{x}_i$ and corresponding labels y_i as their input and optimize over a dense vector $\mathbf{w}$ (and the bias b). In this work we strive to combine the flexibility of kernels with the computational efficiency of linear SVMs. In addition we aim at integrating virtually computed examples that were often shown to amend performance (in, e.g., digit recognition). This naturally requires code changes in the core of the participating SVM solvers and it is not obvious how to achieve this goal with the above considered linear SVM solvers. However, all of the above SVM solvers do not require direct access to elements of $\mathbf{w}$ or examples $\mathbf{x}_i$, but merely require the following two operations:

(i) dot product between feature vector and the vector $\mathbf{w}$: $r \leftarrow \langle \mathbf{x}, \mathbf{w} \rangle$ **DOT**

(ii) multiplication with a scalar $\alpha \in \mathbb{R}$ and addition to the vector $\mathbf{v} \in \mathbb{R}^n$: $\mathbf{v} \leftarrow \alpha \mathbf{x} + \mathbf{v}$ **ADD**

To see this, recall that all SVM solvers mentioned above access the examples only to compute (i) outputs $f(\mathbf{x}_i) = \langle \mathbf{w}, \mathbf{x}_i \rangle$, $i = 1, \ldots, m$, and (ii) a sub-gradient $\mathbf{g} = \sum_{i \in I \subseteq \{1,\ldots,m\}} \pi_i \mathbf{x}_i$, $\pi_i \in \{-1, 0, +1\}$ of $F(\mathbf{w})$ (BMRM, SVM^{perf}, OCAS) or its stochastic estimate (SGD, Pegasos). Note that the Hessian required by the quasi-Newton method (subBFGS) is also estimated from the

sub-gradients. In turn, **DOT** and **ADD** are sufficient operations, i.e., the only operations directly accessing the examples.

We propose a new training framework COFFIN whose main essence is the on demand computation of features and examples within the **DOT** and **ADD** operations. We show that by well organizing these computations the proposed strategy can significantly save both memory and computational demand which are the main hurdles in large-scale learning.

We describe two directions of the framework by introducing kernel like non-linearity (Section 2.1) and learning invariant classifiers using virtual examples (Section 2.2).

2.1 A Kernel Framework

To enable the use of non-vectorial data and kernels, we now consider $(\mathbf{z}_i, y_i)_{i=1}^m \in (\mathcal{Z} \times \{-1, +1\})^m$ as input and a feature extractor $\Phi : \mathcal{Z} \mapsto \mathbb{R}^n$ that maps objects $\mathbf{z}$ from the input space $\mathcal{Z}$ to a real valued vectorial representation. While classical SVM optimizers operating in the dual can easily make use of the kernel trick $k(\mathbf{z}, \mathbf{z}') = \langle \Phi(\mathbf{z}), \Phi(\mathbf{z}') \rangle$, i.e., work without ever *explicitly* computing the mapping $\Phi(\mathbf{z})$, this is not straight-forward in the primal. Even though optimization over kernel expansion provides this trick also in the primal, it again leads to the curse of support vectors and hurts any large-scale learning applications. To endow kernel like non-linearity Φ is commonly applied in a pre-processing step. However, if $\dim(\mathcal{Z}) \ll n$ this quickly renders any kind of large-scale learning infeasible, since only few vectors $\mathbf{x}$ will fit in memory. In addition, it should be noted that such large objects will cause CPU cache misses whenever they are accessed slowing computations down significantly.

Computing Features *on-demand* We propose to use *on-demand* computed features, i.e., instead of applying the mapping $\Phi(\mathbf{z})$ in a preprocessing step we compute the *non-zero elements* of $\mathbf{x} := \Phi(\mathbf{z})$ on demand whenever $\mathbf{x}$ is accessed. Formally, we define the non-zero elements

$$\Phi_{\neq 0}(\mathbf{z}) := \left\{ (\Phi_{\neq 0}(\mathbf{z}))_{v_1}, \ldots, (\Phi_{\neq 0}(\mathbf{z}))_{v_\ell} \right\}$$

and their number as follows

$$|\Phi_{\neq 0}(\mathbf{z})| := \sum_{k=1}^{n} I(\Phi(\mathbf{x})_k \neq 0)$$

where $I(.)$ is the indicator function that evaluates to 1 if its argument is true and to zero otherwise.

For the operations **ADD** and **DOT** to be efficient, it is required that (a) the individual features $\Phi_{\neq 0}(\mathbf{z})_{v_i}$ can be computed quickly, e.g. in $\mathcal{O}(1)$ (b) can be indexed by $v_i,\ i = 1, \ldots, \ell$, (c) their subsequent access to $(\mathbf{w})_{v_i}$ is fast and (d) the number of non-zero features $|\Phi_{\neq 0}(\mathbf{z})|$ is low, i.e., optimally linear in the dimensions of the input

$$\mathcal{O}(|\Phi_{\neq 0}(\mathbf{z})|) = \mathcal{O}(\dim(\mathcal{Z})).$$

Examples for Φ are the construction of a (low-degree) polynomial kernel feature space on very sparse features, string kernel based features (n-gram counts), hashed feature values, decompression algorithms. They are described in more more detail in Section 2.1.2. We now discuss data structures that allow us to efficiently represent $\mathbf{w}$.

Table 1: Computational complexity of the **ADD** and **DOT** operations computed for a single $\mathbf{z}$ for the different data structures. In addition, the memory requirement of $\mathbf{w}$ is shown.

	Dense	Sorted Array	Tree
Add	$\mathcal{O}(\|\Phi_{\neq 0}(\mathbf{z})\|)$	$\mathcal{O}(\|\mathbf{w}\|_{\neq 0}) + \|\Phi_{\neq 0}(\mathbf{z})\|)$	$\mathcal{O}(\Phi_{\neq 0}(\mathbf{z})\|)$ to $\mathcal{O}(d\Phi_{\neq 0}(\mathbf{z})\|)$
Dot	$\mathcal{O}(\|\Phi_{\neq 0}(\mathbf{z})\|)$	$\mathcal{O}(\|\mathbf{w}\|_{\neq 0}) + \|\Phi_{\neq 0}(\mathbf{z})\|)$	$\mathcal{O}(\Phi_{\neq 0}(\mathbf{z})\|)$ to $\mathcal{O}(d\Phi_{\neq 0}(\mathbf{z})\|)$
Mem	$\mathcal{O}(n)$	$\mathcal{O}(\sum_{i=1}^{m} \|\Phi_{\neq 0}(\mathbf{z}_i)\|)$	$\mathcal{O}(\sum_{i=1}^{m} \|\Phi_{\neq 0}(\mathbf{z}_i)\|)$

2.1.1 Data structures for representing $\mathbf{w}$

Dealing with a potentially huge number of features, most of which potentially zero, requires an efficient representation of the SVM-$\mathbf{w}$. Sonnenburg et al. [2007a] noted that there are three basic operations required when dealing with $\mathbf{w}$, a `clear` operation to set all components of $\mathbf{w}$ to zero, an `add` operation that coincides with operation **ADD** in this paper and a `lookup` operation to access all non-zero elements efficiently and is required in the **DOT** operation. We can thus make use of their `linadd` trick to represent the SVM-$\mathbf{w}$ not necessarily as a dense vector, but if more appropriate in a sparse data structure like a tree or a sorted list. In addition, we will make use of hashing to lower the effective number of dimensions. Hashing has been first investigated in depth and successfully used in hash kernels Shi et al. [2009].

We briefly review the data structures and their complexity:

Representing $\mathbf{w}$ as dense vector If n is not overly large then one should keep the whole vector $\mathbf{w}$ in memory. In this case each **ADD** and **DOT** operation can be done in $\mathcal{O}(|\Phi_{\neq 0}(\mathbf{z})|)$ time at a cost of a potentially huge, i.e., $\mathcal{O}(n)$ memory requirement. However, note that the dimensionality of $\mathbf{w}$ is independent of the number of examples m.

Sorted Array More memory efficient considering sparse data, but computationally more expensive are sorted arrays of index-value pairs $\{(v_1, w_{v_1}), \ldots, (v_\ell, w_{v_\ell})\}$. Assuming ordered tuples $(v'_k, (\Phi(\mathbf{x}))_{v'_k}),\ k = 1 \ldots, \ell'$ (indexed by $\mathbf{v}'_k$) **ADD** and **DOT** can be performed in $\mathcal{O}(\ell' + \ell)$.

Tree In particular when dealing with strings a way of organizing non-zero elements are trees, like binary trees, *tries* or suffix arrays Knuth [1973], Fredkin [1960], Teo and Vishwanathan [2006]. Depending on the tree used, the **ADD** operation needs $\mathcal{O}(d|\Phi_{\neq 0}(\mathbf{z})|)$ (trie; d is the depth of the tree) or $\mathcal{O}(|\Phi_{\neq 0}(\mathbf{z})|)$ (suffix array). Similar the complexity of **DOT** varies from $\mathcal{O}(d|\Phi_{\neq 0}(\mathbf{z})|)$ (trie) or $\mathcal{O}(|\Phi_{\neq 0}(\mathbf{z})|)$ (suffix array). Note that the computational complexity of both operations is independent of the number of d-mers/elements stored in the tree but comes at the cost of an additional storage overhead.

Hashing Table 1 summarizes computational complexity and memory requirements of the considered data structures in big-O notation. This unfortunately hides the large constants involved when dealing with the seemingly efficient trees. For example while $\mathcal{O}(\sum_{i=1}^{m} |\Phi_{\neq 0}(\mathbf{z}_i)|)$ seems like a low memory requirement (this quantity is linear in the amount of data), it is sufficient to already impose

practical limits, e.g., Sonnenburg et al. [2007a] require 20 bytes per node for their already tuned DNA-tries; the highly memory-efficient suffix array algorithm of Teo and Vishwanathan [2006] still requires 19 bytes per character. The sorted array has an additional index attached to it increasing data size by at least factor 2.

On the other hand, **DOT** and **ADD** are very fast for dense $\mathbf{w}$ (no hidden large constants) but suffer from huge memory requirements (for some string kernels $n > 10^{14} \gg m$).

This is where hashing comes as a rescue: For an index set J, a number of bits γ, a hash function $h(J) \mapsto 1, \ldots, 2^\gamma$ computes an approximation of $\Phi(\mathbf{z})_i$ via

$$(\widehat{\Phi}(\mathbf{z}))_j = \sum_{i \in J; h(i)=j} (\Phi(\mathbf{z}))_i$$

ignoring potential hash collisions, i.e., the new vector $\widehat{\Phi}(\mathbf{z})$ has only 2^γ dimensions. This trick and the resulting (minor) information loss has been extensively discussed in Shi et al. [2009] w.r.t. both theory showing its influence on generalization bounds and empirically for dense $\mathbf{w}$. It has the big advantage that we can use a fixed hash-table size of size $n = 2^\gamma$ for $\mathbf{w}$ either in dense representation Shi et al. [2009] or a sparse one Sonnenburg et al. [2007a].

It should be noted that the use of such data structures is not exclusive either-or, for example for a string kernel one might want to use a dense representation for short string lengths and for the remaining use sorted arrays, suffix arrays or hashes.

In this work we will exclusively focus on either explicit or hashed dense representations of $\mathbf{w}$ since — for very large m — they have the lowest memory requirements and **DOT** and **ADD** can be computed fastest.

2.1.2 Computing Φ for a Variety of Kernels

In this section we give examples on how to efficiently compute Φ for a variety of kernels.

Polynomial Kernel of low degree The homogeneous[1] polynomial kernel of degree d is defined as $K(\mathbf{z}, \mathbf{z}') = (\langle \mathbf{z}, \mathbf{z}' \rangle)^d$, $\mathbf{z}, \mathbf{z}' \in \mathbb{R}^p$. The feature space of the polynomial kernel can be defined as $\Phi \colon \mathbb{R}^p \to \mathbb{R}^n$

$$\Phi(\mathbf{z}) = \left(\binom{d}{u}^{\frac{1}{2}} \mathbf{z}^{|\mathbf{u}|} \mid \mathbf{u} \in \mathbb{N}^p, |\mathbf{u}| = d \right),$$

where $\mathbf{u} = (u_1, \ldots, u_p) \in \mathbb{N}^p$ is a multi-index, $|\mathbf{u}| = \sum_{i=1}^p u_i$, $\binom{d}{u} = \frac{d!}{(d-|\mathbf{u}|)! \prod_{i=1}^p u_i!}$ and $\mathbf{z}^{|\mathbf{u}|} = \prod_{i=1}^p z_i^{u_i}$. The dimensionality of the feature space is $n(p,d) = \sum_{\mathbf{u} \in \mathbb{N}^p} I(|\mathbf{u}| = d)$. In turn, computing **ADD** and **DOT** operations require in general case $\mathcal{O}(n(p,d))$ operations and thus are feasible only if degree d is low and the input vectors $\mathbf{z}$ are low-dimensional or very sparse. Let $J_{=0}(\mathbf{z})$ be a set of indices of zero components of $\mathbf{z}$ and let $U_{\neq 0}(\mathbf{z}) = \{\mathbf{u} \in \mathbb{N}^p \mid |\mathbf{u}| = d, u_i = 0, i \in J_{=0}(\mathbf{z})\}$ be set of multi-indexes of non-zero monomials. Then, computing **ADD** and **DOT** require traversing only through the non-zero monomials $\mathbf{z}^{|\mathbf{u}|}$, $\mathbf{u} \in U_{\neq 0}(\mathbf{z})$. Hence the computation complexity of sparse **ADD** and **DOT** decreases to $\mathcal{O}(n(p - |J_{=0}(\mathbf{z})|, d))$. Note that one may save memory by using a hashed approximation of the multi-index $h(\mathbf{u})$.

[1] The derivation for the inhomogeneous case is analogous.

Bag of Words, Spectrum and n-gram Kernel The spectrum kernel (e.g. Sonnenburg et al. [2007a]) implements the n-gram or bag-of-words kernel as originally defined for text classification in the context of biological sequence analysis. $\Phi_d(\mathbf{z})$ computes counts of all possible d-gram that are contained in the string $\mathbf{z}$, i.e., given an alphabet Σ and all possible d-grams $\mathbf{u} \in \Sigma^d$

$$\Phi_d(\mathbf{z}) = (\#\mathbf{u}_1(\mathbf{z}), \ldots, \#\mathbf{u}_{|\Sigma^d|}(\mathbf{z})).$$

A flavor of this kernel considers all k-grams of length $1 \ldots d$, i.e.

$$\Phi_d^{wspec}(\mathbf{z}) = (\sqrt{\beta_1}\Phi_1(\mathbf{z}), \ldots, \sqrt{\beta_d}\Phi_d(\mathbf{z})),$$

where $\beta_k \in \mathbb{R}^+$ are some non-negative weights.

For small alphabets and d-gram lengths individual d-grams can be stored in fixed-size variables, e.g., DNA d-gram of length $d \leq 8$ can be efficiently represented as a 16-bit integer values. The ability to store d-gram in fixed-bit variables or even CPU registers greatly improves performance, as only a single CPU instruction is necessary to compare or index a d-gram. The computational complexity of computing Φ is linear in the length of the sequences, i.e., $\mathcal{O}(|\mathbf{z}|_{\neq 0})$. Note that this representation allows efficient computation of the *Weighted Spectrum kernel* in $\mathcal{O}(d|\mathbf{z}|_{\neq 0})$ *without* requiring additional storage. Finally, note that for long strings $\mathbf{z}$ or low d it can indeed be more efficient to store pre-processed tuples of $(\#\mathbf{u} \in \mathbf{z}, \mathbf{u})_{\forall \mathbf{u} \in \mathbf{z}}$ instead.

Depending on the alphabet size, the spectrum kernel is best represented explicitly, i.e., using a dense $\mathbf{w}$, index $\mathbf{u}$ for small alphabets or a dense $\mathbf{w}$ Sonnenburg et al. [2007a], but hashed index $h(\mathbf{u})$ Shi et al. [2009].

Weighted Degree Kernel The *weighted degree* kernel [Sonnenburg et al., 2007a] (WDK) can be conceived as a weighted spectrum kernel for each sequence position. This kernel has been excessively used to detect genomic signals Sonnenburg et al. [2008] and its feature space can be expressed as

$\Phi_d^{wd}(\mathbf{z}) = (\Phi_d^{wspec}(\mathbf{z}_{1,\ldots,d}), \ldots, \Phi_d^{wspec}(\mathbf{z}_{|\mathbf{z}|-d+1,\ldots,|\mathbf{z}|}))$.

As a result the feature space of the WDK is $\mathcal{O}(l|\Sigma|^d)$ dimensional Sonnenburg et al. [2007a] and thus for the usually considered $d = 20$ even for relatively short DNA sequences too big for a dense representation. Naturally Sonnenburg et al. [2007a] used a sparse trie representation for Φ. However, we instead propose to use multiple dense hash tables, one for each degree and position. For this to be efficient we require incremental hashes, i.e. hashes that can be seeded with the previous seed,

$$h(x_{1,\ldots,k+1}, \sigma) = h(x_{1,\ldots,k+1}, h(x_{1,\ldots,k}, (\ldots(h(x_1, \sigma)))),$$

where σ is some initial seed.

Similar to the spectrum kernel we can explicitly represent k-grams for small k to further speed up computations.

Other Examples Other examples for Φ include fast decompression algorithms like LZO[2] that can efficiently decompress a sequence at one third of the speed of a usual `memcpy`, but also other expert

[2] `http://www.oberhumer.com/opensource/lzo/`

chosen general basis functions like sine waves, exponentials etc. While one could use the empirical kernel map or sparse kernel approximations to approximate general kernels the kernel evaluations with a subset of the training examples creates a huge speed penalty rendering on-the-fly computation of Φ hard.

2.2 Incorporating Invariance by Virtual Examples

In many applications, we know that there are transformations of the input measurements $\mathbf{z} \in \mathcal{Z}$ which leave the class membership y invariant. A commonly used way to incorporate prior knowledge into SVM classifiers is to augment the set of training examples with virtual examples (VE) that are created by applying a set of transformations (against which we want invariance) to the training examples DeCoste and Schölkopf [2002].

To put it formally, our prior knowledge is described by a set $\mathcal{T}$ which contains a finite number of transformations $T\colon \mathcal{Z} \mapsto \mathcal{Z}$. We require that

$$f(\Phi(T\mathbf{z}_i)) = y_i \ , \qquad \forall T \in \mathcal{T}, i = 1, \ldots, m \ ,$$

where $\{(\mathbf{z}_i, y_i)\}_{i=1}^m$ are given training examples. Training of f can be expressed as training of a standard SVM classifier from $|\mathcal{T}|m$ virtual training examples

$$\{(\mathbf{z}, y) \mid \mathbf{z} = T\mathbf{z}_i, y = y_i, T \in \mathcal{T}, i = 1, \ldots, m, \} \ .$$

The VE method has two important advantages. First, it does not impose any constraints on the transformations $\mathcal{T}$. Second, existing SVM solvers can be used to train the invariant classifier. However, the cardinality of $\mathcal{T}$ may increase exponentially when the transformation T is composed of s simpler ones, $T = T_1 \odot \cdots \odot T_s$ and thus $\mathcal{T} = \mathcal{T}_1 \times \cdots \times \mathcal{T}_s$. Thus, VE are computationally demanding because they (a) significantly increases the number of training examples and (b) pose huge memory requirements to store all $m|\mathcal{T}|$ virtual examples.

This is were COFFIN comes as a rescue: Instead of pre-computing the VE in advance, we generating them *on demand*. Since only the original examples need to be stored in memory, this approach drastically reduces memory requirements. In case when transformations T can be computed quickly, the on demand generation of the virtual examples also speeds up the training. E.g., transformations of 2D images (needed in OCR and image recognition) can be computed on GPUs — a dedicated hardware for these transformations.

In Section 3.2, we demonstrate effectiveness of the proposed approach COFFIN on the problem of gender estimation from face images showing that COFFIN has by an order of magnitude less memory requirements compared to the standard approach. A practical outcome is that we can train the gender classifier from $4,808,250$ example images on a notebook with only 4GB of memory instead of high-end computing node with > 50GB of memory.

2.3 Implementation and Parallelization

We integrated COFFIN in the state-of-the-art cutting plane solver OCAS, the dual coordinate descent based LibLinear and SGD. We implemented a general framework that allows stacking of a variety of features that support **ADD** and **DOT** operations, namely dense and sparse real-valued features,

weighted spectrum and WD kernel features for specified k-mer length, once using an explicit representation and once using hashing. We implemented the virtual example method to OCAS solver as its API provides easy way to customize **ADD** and **DOT** operations. A pointer to our implementation will be disclosed upon acceptance of the paper.

Since the **DOT** operation is the most time consuming when performing predictions and when using batch-solvers we trivially parallelized this part of the code (based on shared memory parallelization, i.e., posix threads). However, an important detail here needs considerable attention: *memory locality*. CPUs nowadays are i/o bound, i.e. computation speed is drastically limited by memory speed and parallelized code cannot help this. To alleviate that bottleneck, off the shelf CPUs have rather large data and instruction caches. For example an AMD Opteron CPUs often have 64k level 1 data cache and 1MB level 2 data cache.[3] Within the **DOT** operation, it is highly beneficial to split $\mathbf{w} = (\mathbf{w}_{B_1}, \ldots, \mathbf{w}_{B_k})$ into smaller blocks, parallelizing within each block $\mathbf{r} = \sum_{j=1}^{k} \sum_{i=1}^{t} \langle \mathbf{x}_{i,B_j}, \mathbf{w}_{B_j} \rangle \mathbf{e}_i$ where the inner sum is distributed among cores.

3 Experiments

3.1 Human Acceptor Splice Site Recognition

To demonstrate the effectiveness of our proposed method COFFIN , we apply it to the problem of human acceptor splice site detection. This problem can be formulated as a two-class classification problem discriminating true splice sites from fake ones. Due to the importance of this problem in computational gene finding, many different methods to detect splice sites have been proposed. They all predict splice sites based on the local context, i.e., a short window around the actual splice site. Currently, support vector machines are the most accurate splice site detectors Degroeve et al. [2005], Sonnenburg et al. [2007b], Franc and Sonnenburg [2009]. In particular, Sonnenburg et al. [2007b] showed that prediction accuracy steadily increases with training sample size. However, even though they already used the `linadd` algorithm Sonnenburg et al. [2007a] to speed up string kernel-based SVMs on a quad-core system, they could not use all available 50 million training points (but "only" 8 million). Sonnenburg et al. [2007b] achieved 54.42% area under the precision recall curve (auPRC, Davis and Goadrich [2006]) in a genome-wide study on human acceptor splice sites.

On the other hand, Degroeve et al. [2005] trained a linear SVM based on a number of preselected and explicitly computed string kernel feature spaces that are subsets from the spectrum and WD kernel feature spaces: Left and right of the splice site spectrum kernels of order 3 up to order 6 were used. Over the whole window, a WD kernel of order 3 with weights equal to 1 was used. Even though this approach scales well, they used $< 100,000$ data points (potentially, since they relied on the unmodified SVM$^{\text{light}}$ binary).

Recently, Franc and Sonnenburg [2009] demonstrated in a proof of concept study for OCAS that training on all the available examples improves performance. However, they could not use the full potential of higher order string kernels and achieve inferior performance compared to Sonnenburg et al. [2007b] for the same sample size (cf., Ocas $d = 8$ vs. Linadd $d = 20$ in Table 2).

[3] `http://en.wikipedia.org/wiki/Opteron`

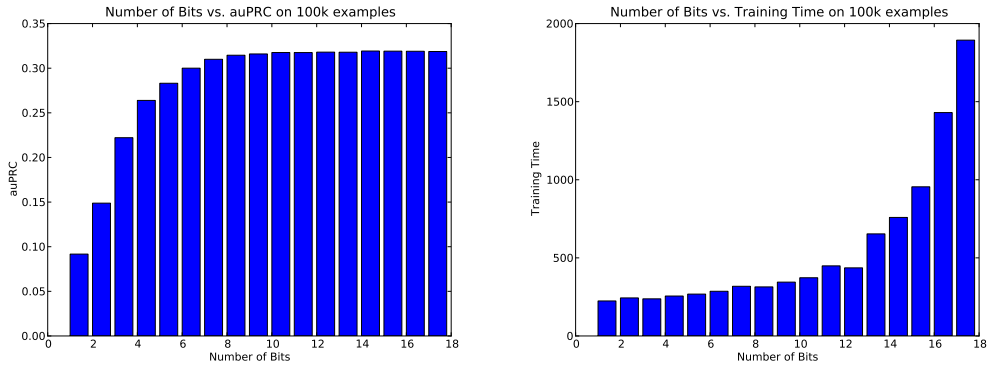


Figure 1: Performance in terms of auPRCand training times on the human acceptor splice site experiment using 100,000 examples and varying bit sizes for the hash of the central WD kernel. For this experiment OCAS was used. It can be seen that already starting with about 8-10 bits the auPRCreaches a plateau. In addition, training times start to drastically increase as soon as hashes of more than 12 bit are used. This drop in performance indicates that the whole hash-table does no longer fit in the CPU data cache for larger hash tables. For 12bits $\mathbf{w}$ is 11,725,480 dimensional.

Experimental Setup Following Franc and Sonnenburg [2009], we train COFFIN on all available 50 million strings of length 141 using the features corresponding to two weighted spectrum kernels (one left and one right of the splice site, i.e., positions 1-59 and 62-141) and a WD kernel. We fixed $C = 1$ and used the weighted spectrum kernel order $d = 8$, for the WD kernel order $d = 8$ or $d = 20$ respectively, which were found optimal in Franc and Sonnenburg [2009]. We use a dense explicit $174,760$ dimensional representation for the spectrum kernels and dense or hashed representations for the WD kernel for hash sizes $\gamma = 12$ and $\gamma = 16$ (cf., Fig. 1 for a discussion about optimal hash sizes). The resulting spanned string kernel feature space has $12,495,340$ (WD $d = 8$ explicit), $11,725,480$ (WD $d = 12$ hash $\gamma = 16$) and $184,986,280$ (WD $d = 20$ hash $\gamma = 16$) dimensions respectively.

As the raw string-based dataset is already of size $7.1 \cdot 10^9$ bytes and even a sparse representation of each string increases the dataset by a factor of more than 3,000 ($(141 + 59 + 80) \cdot 12$ bytes per feature vector, assuming a 4 byte integer and an 8 byte float), it is only by the means of COFFIN that we can solve such huge optimization problems.

To provide a fair comparison we measure training times and auPRC on a held out test set of 4,627,840 examples for various training set sizes using low order WD kernels and explicit representation or higher order ones with hashing. We consider SVM$^{\text{light}}$ employing `linadd` and OCAS and liblinear employing COFFIN in this comparison.[4]

Using COFFIN within liblinear, training on 50 million examples using a single CPU-Core of a 16 core AMD Opteron Linux-based machine, leads to record area under the precision recall curve (auPRC) of 58.57% in less than 3 days. For comparison, the previous best dual method already using the `linadd` speedups Sonnenburg et al. [2007a] achieved an auPRC of 54.42% training on just $8 \cdot 10^6$ examples in about 11 days. Franc and Sonnenburg [2009] achieved an auPRC of 57.77%

[4]Preliminary results showed that SGD failed miserably — potentially requiring step-size tuning for the different d.

Table 2: Training times and auPRCfor human splice site detection for various data set sizes and $\mathbf{w}$ representations and $d's$ of the weighted degree kernel. (first row) The previous state-of-the-art was an SVM^{light} employing `linadd` and a weighted degree shift kernel; numbers are estimated from Figure 2 in Sonnenburg et al. [2007b]. The result marked* was extrapolated (linearly scaled). COFFIN employing OCAS (OC) and liblinear (LL) is compared to `linadd` and by far outperforms `linadd` in accuracy and speed when using hashing. Using a dense representation is even slower than hashing and has inferior performance.

Method / Sample Size	10^4		10^5		10^6		10^7		$5 \cdot 10^7$	
SVM^{light}+`linadd` WDS	$\approx$ 25s	$\approx 11\%$	$\approx$ 500s	$\approx 28\%$	$\approx 2 \cdot 10^5$s	$\approx 44\%$	$\approx 10^6$	$\approx 54\%$	-	
SVM^{light}+`linadd` $d = 8$	57s	10.37%	970s	28.62%	34110s	43.78%	-		-	
SVM^{light}+`linadd` $d = 20$	56s	11.15%	1033s	31.80%	34586s	46.27%	-		-	
COFFIN OC $d = 8$ (dense)	167s	10.00%	948s	28.57%	9952s	43.84%	131202s	53.26%	656010s*	57.77%
COFFIN OC $d = 20, \gamma = 12$	65s	10.81%	435s	31.80%	5349s	46.12%	76311s	54.31%	908654s	57.89%
COFFIN OC $d = 20, \gamma = 16$	363s	10.66%	1430s	31.90%	10288s	46.43%	-		-	
COFFIN LL $d = 20, \gamma = 12$	61s	10.59%	360s	31.59%	3783s	45.97%	25902s	54.17%	132581s	57.75%
COFFIN LL $d = 20, \gamma = 16$	111s	10.52%	604s	31.69%	4590s	46.26%	44232s	54.56%	247907s	58.57%

using OCAS — we obtain the same precision in *one fifth* of the time (cf., Table 2). We could not train OCAS on the $\gamma = 16$ hashed data set since a single cutting plane already requires about $1.4GB$ of memory. However, since OCAS is a batch method, we could train it using 16 CPU cores on 50 million examples employing parallelized **DOT** operation resulting in 19hours spend in computing outputs instead of 252 (speedup factor 13) demonstrating the effectiveness of our memory access pattern. Since liblinear is an online style solver, it cannot easily be parallelized but could benefit from the recent work of M. Zinkevich [2009] on delayed gradient updates. Liblinear with $d = 20$, $\gamma = 16$ involves a feature space of size 184,986,281. Training times for liblinear using L2 loss were slightly lower at the cost of slightly decreased performance (results not shown).

3.2 Gender Estimation From Face Images

The task of this binary classification problem is to discriminate digital images into two classes – males and females. A robust classifier should be invariant against common image transformations: translation, rotation, scale and illumination changes. Currently, there exist feature representations invariant only against one of the transformations. For example Local Binary Patterns (LBP) Ojala et al. [2002] are the current state-of-the-art image descriptors thare are invariant against any monotonic change in intensity values. In order to gain robustness against translation, rotation and scale, we apply the method of virtual examples (c.f. Sec. 2.2) in training. As mentioned, the bottle neck of the method is that the set of virtual examples quickly grows very large, imposing artificial memory limits.We show that using COFFIN computing virtual examples on demand significantly alleviates the memory problem at the price of only minor increase in training time. In particular, we train the gender classifier from 4,808,250 virtual examples on a notebook with Intel 2.66 GHz CPU and only 4GB of memory. Storing all the precomputed examples in memory would require more than 50GB.

We collected a dataset of 18,504 images with human faces downloaded from the Internet. We split the images into 12,822 training and 5,682 test examples. The faces were manually segmented and labeled with the gender. The segmentation is given by a position and size of a window containing a face. When applying the classifier in "real world", position and size of the window are taken from

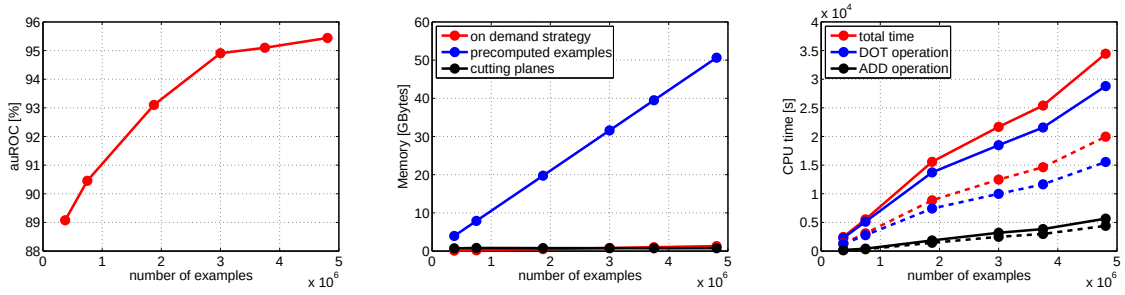


Figure 2: Performance of the gender classifier trained from virtual examples. Figure (a) shows auROC w.r.t. training set size. In figure (b), we plot memory requirements of (i) the proposed on demand training strategy (red) and (ii) the standard strategy based on pre-computing all examples (blue). The black curve shows extra memory needed by OCAS solver to store the cutting planes (black) which is common for both strategies. Figure (c) shows CPU time needed by **DOT** operation (blue), **ADD** operation (black) and the total time (red). The solid lines correspond to the on demand computation strategy while the curves for standard approach are dashed.

a pre-trained face detector and are thus often are imprecise. To cope with this situation, we cropped the testing faces using the annotated segmentation applying translational and rotational distortions: The position distortion was in range of ± 2 pixels in both axis and the distortion in scale was $\pm 5\%$ of the base window size. The parameters of the distortions were estimated from outputs of a real AdaBoost face detector.

We generate the virtual examples by applying the mentioned transforms to the annotated images. The classifier input is a quadruple $\mathbf{x} = (\mathbf{I}, \mathbf{p}, s, \varphi)$, where $\mathbf{I} \in \mathbb{N}^{90\times 60}$ is gray-scale image, $\mathbf{p} \in \mathbb{N}^2$ is position of the 60×40 pixels base window, $s \in \mathbb{R}$ is scale and $\varphi \in \mathbb{R}$ is the rotation of the base window. We define the transformation

$$T(\Delta\mathbf{p}, \Delta s, \Delta\varphi)(\mathbf{I}, \mathbf{p}, s, \varphi) = (\mathbf{I}, \mathbf{p} + \Delta\mathbf{p}, s + \Delta s, \varphi + \Delta\varphi) \ ,$$

parametrized by the triplet $(\Delta\mathbf{p}, \Delta s, \Delta\varphi)$ which defines the change in translation, scale and rotation, respectively. Then we construct the set $\mathcal{T} = \{T(\Delta\mathbf{p}, \Delta s, \Delta\varphi) \mid \Delta p \in \mathcal{T}_p, \Delta s \in \mathcal{T}_s, \Delta\varphi \in \mathcal{T}_\varphi\}$ where

$$\begin{aligned} \mathcal{T}_p &= \{\Delta\mathbf{p} \mid \Delta\mathbf{p} = (u; v), u, v \in \{-2, -1, 0, 1, 2\}\} \\ \mathcal{T}_s &= \{\Delta s \mid s \in \{-0.05, 0, 0.05\}\} \\ \mathcal{T}_\varphi &= \{\Delta\varphi \mid \Delta\varphi \in \{-6, -3, 0, 3, 6\}\} \end{aligned}$$

i.e., for each training image we generate $|\mathcal{T}| = 5^2 \cdot 5 \cdot 3 = 357$ virtual examples.

The feature representation $\Phi(\mathbf{I}, \mathbf{p}, s, \varphi) \in \mathbb{R}^n$ is computed from responses of the LPB filter on pyramidal representation of the base window that is cropped from image $\mathbf{I}$ according to $(\mathbf{p}, s, \varphi)$. The pyramid is composed of images 60×40, 30×20, 15×10, 7×5 pixels, obtained from the base window which was then three times downscaled by factor 2. It results in $n = 723,712$-dimensional sparse feature vector composed of all zeros and $2,827$ coordinates equal to one. The feature vector is most efficiently represented by storing $2,827$ indexes (4 bytes each) of non-zero coordinates, i.e., we need $2,827 \cdot 4 = 11,308$ bytes per example. Hence pre-computing all $357 \cdot 12,822 = 4,808,250$ virtual examples requires ≈ 51 GB.

We alleviate the memory problem by computing the virtual examples on demand. For each training image, we pre-compute only their rotated and scaled versions because these are the most expensive operations and they still fit to memory; to store $12,822 \cdot 5$ (rotation) $\cdot 3$ (scale) $= 192,330$ images we need 1.3 GB. The image translations and the pyramid of LBP features are computed on demand. Note that translating the image requires only reading from corresponding memory space and thus it is fast. Similarly, downscaling image by 2, needed for pyramid computation, involves only summing pairs of neighboring image rows and columns. The resulting sums do not need to be normalized as the LBP representation is invariant to monotonic change of intensities.

For different training set sizes, $m \in 375 \cdot \{1000, 2000, \ldots, 10000, 12822\}$, we trained SVMs using OCAS and measured accuracy, memory requirements and computation time. The results presented further are obtained for $C = 0.001$ which was found by tuning on the original training examples. Figure 2 (a) shows the accuracy, measured in terms of the area under ROC (auROC, Fawcett [2003]), w.r.t. number of examples. It is seen that generating more virtual examples significantly helps the performance. The classifier trained only on the original $12,822$ examples with undistorted annotation has $\mathrm{auROC} = 89.57\%$ compared to $\mathrm{auROC} = 95.44\%$ obtained with all virtual examples. Figure 2 (b) shows the memory requirements w.r.t. training set size for (i) all examples precomputed and (ii) examples computed on demand. Computing examples on demand requires ≈ 40 times less memory. Figure 2(c) shows total time of OCAS, the time for computing outputs (dominated by **DOT** operation) and the time for computing cutting planes (**ADD** operation). Because all precomputed examples do not fit into 4GB RAM, the time **ADD** and **DOT** operations is estimated from r_{add} and r_{dot}. However, memory efficiency comes at the cost of slightly slower training, i.e., **ADD** (**DOT**) is on average $r_{add} = 1.28$ ($r_{dot} = 1.85$) times slower. Considering the increase in performance by being able to train on almost 5 million examples the slight decrease in training time (on average 1.75 times slower) seems negligible. In addition, a principled caching strategy might bring speed up to par.

4 Conclusion

We have presented COFFIN — a very efficient computational framework for on-demand creation of features and virtual examples. COFFIN combines the computational efficiency of linear SVMs with the flexibility of kernel based learning. In the experimental section we have demonstrated (a) that our approach allows efficient computations even on low-budget hardware and (b) leads to state-of-the-art results solely due to the fact that enables the use of all available training data. For example we could train a linear SVM on about 5 million example images for the task of gender recognition on a standard notebook with just 4GB of memory and a linear SVM for human acceptor splice site recognition on 50 million examples in a more than 184 million dimensional feature space in less than 3 days achieving new record performance. Still, we see further potential in improving our approach, by considering the computational costs to create virtual vectors $T(\mathbf{x})$ or features $\Phi(\mathbf{z})$ respectively in the core optimization process of the linear SVM solver. For example computed elements could be cached and solvers could put focus on optimizing for the few cached elements before tuning the rest.

References

L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *NIPS 20*. MIT Press, 2008.

R. Collobert, F. Sinz, J. Weston, and L. Bottou. Trading convexity for scalability. In *ICML*, pages 201–208, 2006.

J. Davis and M. Goadrich. The relationship between precision-recall and roc curves. In *ICML*, 2006.

D. DeCoste and B. Schölkopf. Training invariant support vector machines. *Machine Learning*, 46: 161–190, 2002.

S. Degroeve, Y. Saeys, P. De Baets, B. Rouzé, and Y. Van de Peer. SpliceMachine: predicting splice sites from high-dimensional local context representations. *Bioinformatics*, 21(8):1332–8, 2005.

R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *JMLR*, 9:1871–1874, 2008.

T. Fawcett. Roc graphs: Notes and practical considerations for data mining researchers. Technical report hpl-2003-4, HP Laboratories, Palo Alto, CA, USA, January 2003.

V. Franc and S. Sonnenburg. Optimized cutting plane algorithm for large-scale risk minimization. *JMLR*, 2009.

E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

T. Joachims. Training linear svms in linear time. In *KDD'06*, 2006.

S. Keerthi, O. Chapelle, and D. DeCoste. Building support vector machines with reduced classifier complexity. *JMLR*, 7:1493–1515, 2006.

D. E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley, 1973.

J. L. M. Zinkevich, A. Smola. Slow learners are fast. In *NIPS*, 2009.

T. Ojala, M. Pietikäinen, and T. Mäenpää. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE PAMI*, 24(7):971–987, 2002.

B. Schölkopf and A. Smola. *Learning with Kernels*. The MIT Press, MA, 2002.

Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *JMLR*, 10:2615–2637, Nov 2009.

S.-S. Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated sub-gradient solver for svm. In *ICML*, pages 807–814. ACM Press, 2007.

S. Sonnenburg, G. Rätsch, and K. Rieck. Large scale learning with string kernels. In *Large Scale Kernel Machines*. MIT Press, 2007a.

S. Sonnenburg, G. Schweikert, P. Philips, J. Behr, and G. Rätsch. Accurate Splice Site Prediction. *BMC Bioinformatics*, 8:(Suppl. 10):S7, December 2007b.

S. Sonnenburg, A. Zien, P. Philips, and G. Rätsch. Positional oligomer importance matrices. *Bioinformatics*, July 2008.

I. Steinwart. Sparseness of support vector machines – some asymptotically sharp bounds. In *Proceedings of NIPS Conference*, pages 169–184, 2004.

C.-H. Teo and S. V. N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proc. 23rd ICMP*, pages 939–936. ACM Press, 2006.

C. H. Teo, Q. Le, A. Smola, and S. Vishwanathan. A scalable modular convex solver for regularized risk minimization. In *KDD'07*, August 2007.

J. Yu, S. Vishwanathan, S. Günter, and N. Schraudolph. A quasi-newton approach to nonsmooth convex optimization. In *ICML 2008*, 2008.