Journal of Machine Learning Research 11 (2010) 1471-1490 Submitted 8/09; Revised 1/10; Published 4/10

Training and Testing Low-degree Polynomial Data Mappings via Linear SVM

Yin-Wen Chang b92059@csie.ntu.edu.tw
Cho-Jui Hsieh b92085@csie.ntu.edu.tw
Kai-Wei Chang b92084@csie.ntu.edu.tw
Department of Computer Science
National Taiwan University
Taipei 106, Taiwan

Michael Ringgaard ringgaard@google.com
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA

Chih-Jen Lin cjlin@csie.ntu.edu.tw
Department of Computer Science
National Taiwan University
Taipei 106, Taiwan

Editor: Sathiya Keerthi

Abstract

Kernel techniques have long been used in SVM to handle linearly inseparable problems by transforming data to a high dimensional space, but training and testing large data sets is often time consuming. In contrast, we can efficiently train and test much larger data sets using linear SVM without kernels. In this work, we apply fast linear-SVM methods to the explicit form of polynomially mapped data and investigate implementation issues. The approach enjoys fast training and testing, but may sometimes achieve accuracy close to that of using highly nonlinear kernels. Empirical experiments show that the proposed method is useful for certain large-scale data sets. We successfully apply the proposed method to a natural language processing (NLP) application by improving the testing accuracy under some training/testing speed requirements.

Keywords: decomposition methods, low-degree polynomial mapping, kernel functions, support vector machines, dependency parsing, natural language processing

1. Introduction

Support vector machines (SVMs) (Boser et al., 1992; Cortes and Vapnik, 1995) have been popular for data classification. An SVM often maps data to a high dimensional space and then employs kernel techniques. We refer to such an approach as nonlinear SVM. Training nonlinear SVM is usually performed through the use of popular decomposition methods. However, these decomposition approaches require considerable time for large data sets. In addition, the testing procedure is slow due to the kernel calculation involving support vectors and testing instances.

For some applications with data in a rich dimensional space (e.g., document classification), people have shown that testing accuracy is similar with/without a nonlinear mapping. If data are not mapped, recently some methods have been proposed to efficiently train much larger data sets. We refer to such cases as linear SVM.

Among the recent advances in training large linear SVM, Hsieh et al. (2008) discuss decomposition methods for training linear and nonlinear SVM. If l is the number of training data, $\bar{n}$ is the average number of non-zero features per instance, and each kernel evaluation takes $O(\bar{n})$ time, then the cost per decomposition iteration for nonlinear SVM is $O(l\bar{n})$. Taking the property of linear SVM, Hsieh et al.'s approach runs one iteration in only $O(\bar{n})$. If the number of iterations is not significantly more than that for the nonlinear case, their method is very efficient for training linear SVM.

Motivated by the above $O(l\bar{n})$ and $O(\bar{n})$ difference, in this work, we investigate the performance of applying linear-SVM methods to low-degree data mappings. By considering the explicit form of the mapping, we directly train a linear SVM. The cost per decomposition iteration is $O(\hat{n})$, where $\hat{n}$ is the new average number of non-zero elements in the mapped vector. If $\hat{n} < l\bar{n}$, the new strategy may be faster than the training using kernels.

Currently, polynomial kernels are less widely used than the RBF (Gaussian) kernel, which maps data to an infinite dimensional space. This might be because under similar training and testing cost, a polynomial kernel may not give higher accuracy. We show for some data, the testing accuracy of using low-degree polynomial mappings is only slightly worse than RBF, but training/testing via linear-SVM strategies is much faster. Therefore, our approach takes advantages of linear methods, while still preserves a certain degree of nonlinearity. Some early works (e.g., Gertz and Griffin, 2005; Jung et al., 2008; Moh and Buhmann, 2008) have employed this idea in their experiments. Here we aim at a more detailed study on large-scale scenarios.

An exception where polynomial kernels have been popular is NLP (natural language processing). Some have explored the fast calculation of low-degree polynomial kernels to save the testing time (e.g., Isozaki and Kazawa, 2002; Kudo and Matsumoto, 2003; Goldberg and Elhadad, 2008). However, these works still suffer from the slow training because of not applying some recently developed training techniques.

This paper is organized as follows. We introduce SVM in Section 2. In Section 3, we discuss the proposed method for efficiently training and testing SVM for low-degree polynomial data mappings. A particular emphasis is on the degree-2 polynomial mapping. Section 4 presents the empirical studies. We give an NLP application on dependency parsing in Section 5. Conclusions are in Section 6.

Notation: we list some notation related to the number of features.

n: number of features (dimensionality of data); $\boldsymbol{x}_i \in R^n$ is the ith training instance

n_i: number of non-zero feature values of $\boldsymbol{x}_i$

$\bar{n}$: average number of non-zero elements in $\boldsymbol{x}_i$; see (9)

$\hat{n}$: average number of non-zero elements in the mapped vector $\phi(\boldsymbol{x}_i)$

$\tilde{n}$: number of non-zero elements in the weight vector $\boldsymbol{w}$

2. Linear and Nonlinear SVM

Assume training instance-label pairs are $(\boldsymbol{x}_i, y_i)$, $i = 1, \ldots, l$, where $\boldsymbol{x}_i \in R^n$ and $y_i \in \{1, -1\}$. We consider the following SVM problem with a penalty parameter $C > 0$:

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum\nolimits_{i=1}^{l} \max(1 - y_i\boldsymbol{w}^T\phi(\boldsymbol{x}_i), 0). \tag{1}$$

The function $\phi(\boldsymbol{x})$ maps an instance to a higher dimensional space for handling linearly inseparable data. We refer to such a setting as nonlinear SVM. For some applications, $\phi(\boldsymbol{x}) = \boldsymbol{x}$ can already properly separate data; we call such cases linear SVM. Many SVM studies consider $\boldsymbol{w}^T\boldsymbol{x}_i + b$ instead of $\boldsymbol{w}^T\boldsymbol{x}_i$ in (1). In general this bias term b does not affect the performance much, so here we omit it for the simplicity.

Due to the high dimensionality of $\phi(\boldsymbol{x})$ and the possible difficulty of obtaining the explicit form of $\phi(\boldsymbol{x})$, SVM is often solved through the dual problem with the kernel trick:

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T Q\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_i \le C, i = 1, \ldots, l, \end{aligned} \tag{2}$$

where $Q_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) = y_i y_j \phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ and $\boldsymbol{e} = [1, \ldots, 1]^T$. $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is called the kernel function and $\boldsymbol{\alpha}$ is the dual variable.

The matrix Q in the dual problem (2) is dense and may be too large to be stored in the computer memory. Currently, decomposition methods (e.g., Joachims, 1998; Keerthi et al., 2001; Chang and Lin, 2001) are the major approach to solve (2). However, if linear SVM is considered, we can more easily solve both the primal and the dual problems. Early studies (e.g., Mangasarian and Musicant, 1999; Ferris and Munson, 2003) have demonstrated that many traditional optimization methods can be applied. They focus on data with many instances but a small number of features. Recently, an active research topic is to train linear SVM with both large numbers of instances and features (e.g., Joachims, 2006; Shalev-Shwartz et al., 2007; Bottou, 2007; Hsieh et al., 2008; Langford et al., 2009).

3. Using Linear SVM for Low-degree Polynomial Data Mappings

In this section, we discuss the methods and issues in training/testing low-degree data mappings using linear SVM. We are interested in when the training via linear-SVM techniques is faster than nonlinear SVM. We put an emphasis on the degree-2 polynomial mapping.

3.1 Low-degree Polynomial Mappings

A polynomial kernel takes the following form

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + r)^d, \tag{3}$$

where γ and r are parameters and d is the degree. The polynomial kernel is the product between two vectors $\phi(\boldsymbol{x}_i)$ and $\phi(\boldsymbol{x}_j)$. For example, if $d = 2$ and $r = 1$, then

$$\phi(\boldsymbol{x}) = [1, \sqrt{2\gamma}x_1, \ldots, \sqrt{2\gamma}x_n, \gamma x_1^2, \ldots, \gamma x_n^2, \sqrt{2}\gamma x_1x_2, \ldots, \sqrt{2}\gamma x_{n-1}x_n]^T. \tag{4}$$

The coefficient $\sqrt{2}$ in (4) is only used to make $\phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}_j)$ have a simple form. Without using kernels, we can consider more flexible mapping vectors. For example, if $\gamma = 1$, removing $\sqrt{2}$ in (4) results in a simple mapping vector:

$$\phi(\boldsymbol{x}) = [1, x_1, \ldots, x_n, x_1^2, \ldots, x_n^2, x_1x_2, \ldots, x_{n-1}x_n]^T. \tag{5}$$

For the polynomial kernel (3), the dimensionality of $\phi(\boldsymbol{x})$ is

$$C(n+d, d) = \frac{(n+d)(n+d-1)\cdots(n+1)}{d!},$$

which is obtained by counting the number of terms in (3).

3.2 Training by Linear SVM Methods

The training time for SVM depends on the number of data instances and the number of features. Due to the high dimensionality of $\phi(\boldsymbol{x})$, we must judge whether it is better to choose an explicit mapping or an implicit way by kernels. We explore this issue by investigating the difference between applying decomposition methods to solve the dual problem of linear and nonlinear SVM. Though many optimization methods have been applied to train SVM, we discuss decomposition methods because of the following reasons. First, they are the major approach for nonlinear SVM. Second, efficient decomposition methods for linear SVM have been developed (e.g., Hsieh et al., 2008).

A decomposition method iteratively updates a small subset of variables. We consider the situation of updating one variable at a time.[1] If $\boldsymbol{\alpha}$ is the current solution and the ith component is selected for update, then we minimize the following one-variable problem:

$$\begin{aligned} \min_{d} \quad & \frac{1}{2}(\boldsymbol{\alpha} + d\boldsymbol{e}_i)^T Q(\boldsymbol{\alpha} + d\boldsymbol{e}_i) - \boldsymbol{e}^T(\boldsymbol{\alpha} + d\boldsymbol{e}_i) \\ & = \frac{1}{2}Q_{ii}d^2 + (Q\boldsymbol{\alpha} - \boldsymbol{e})_i d + \text{constant} \qquad (6) \\ \text{subject to} \quad & 0 \leq \alpha_i + d \leq C. \end{aligned}$$

This minimization is easy, but to construct (6), we must calculate

$$(Q\boldsymbol{\alpha} - \boldsymbol{e})_i = \sum\nolimits_{j=1}^{l} Q_{ij}\alpha_j - 1 = \sum\nolimits_{j=1}^{l} y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)\alpha_j - 1. \tag{7}$$

If each kernel element costs $O(\bar{n})$, where $\bar{n}$ is the average number of non-zero features, then (7) needs $O(l\bar{n})$ operations.

If using the explicit mapping vectors, we can calculate $(Q\boldsymbol{\alpha} - \boldsymbol{e})_i$ by

$$\sum\nolimits_{j=1}^{l} Q_{ij}\alpha_j - 1 = y_i \boldsymbol{w}^T\phi(\boldsymbol{x}_i) - 1, \text{ where } \boldsymbol{w} = \sum\nolimits_{j=1}^{l} y_j\alpha_j\phi(\boldsymbol{x}_j). \tag{8}$$

If $\boldsymbol{w}$ is available, (8) requires $O(\hat{n})$ operations, where $\hat{n}$ is the average number of non-zero elements in $\phi(\boldsymbol{x}_i)$, $\forall i$. To maintain $\boldsymbol{w}$, Hsieh et al. (2008) use

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_i(\alpha_i^{\text{new}} - \alpha_i^{\text{old}})\phi(\boldsymbol{x}_i),$$

[1] If using standard SVM with the bias b, the dual form contains an equality and at least two variables must be considered.

so the cost is also $O(\hat{n})$. Therefore, the above discussion indicates the tradeoff between $O(l\bar{n})$ and $O(\hat{n})$ cost by implicit and explicit mappings, respectively.

Practical implementations of decomposition methods involve other issues. For example, if using the kernel trick, we may develop better techniques for selecting the working variable at each iteration. Then the number of iterations is smaller. More details can be found in Hsieh et al. (2008, Section 4). Nevertheless, checking $O(l\bar{n})$ and $O(\hat{n})$ can roughly indicate if using an explicit mapping leads to faster training.

3.3 Number of Non-zero Features per Instance

From the discussion in Section 3.2, it is important to know the value $\hat{n}$. If the input data are dense, then the number of non-zero elements in $\phi(\boldsymbol{x})$ is $O(n^d)$, where d is the degree of the polynomial mapping.

If the input data are sparse, the number of non-zero elements is smaller than the dimensionality. Assume n_i is the number of non-zero elements of the ith training instance. Then the average number in $\boldsymbol{x}_i$ $\forall i$ is

$$\bar{n} = \frac{n_1 + \cdots + n_l}{l}. \tag{9}$$

If $d = 2$, the average number of non-zero elements in $\phi(\boldsymbol{x}_i)$, $\forall i$ is

$$\hat{n} = \frac{1}{l}\sum\nolimits_{i=1}^{l} \frac{(n_i+2)(n_i+1)}{2} \approx \frac{1}{l}\sum\nolimits_{i=1}^{l} \frac{n_i^2}{2} = \frac{1}{2}\bar{n}^2 + \frac{1}{2l}\sum\nolimits_{i=1}^{l}(n_i - \bar{n})^2. \tag{10}$$

The second term in (10) is the variance of $n_1, \ldots, n_l$. If the variance is small, comparing $l\bar{n}$ and $\bar{n}^2/2$ can possibly indicate if one should train a linear or a nonlinear SVM. In Section 4, we give more analysis on real data.

3.4 Implementation Issues

Due to the high dimensionality of $\phi(\boldsymbol{x}_i)$, some implementation issues must be addressed. To begin, we discuss various ways to handle the new data $\phi(\boldsymbol{x}_1), \ldots, \phi(\boldsymbol{x}_l)$. A vector $\phi(\boldsymbol{x})$ now contains terms like $x_r x_s$, which can be calculated using $\boldsymbol{x}$. We consider three methods:

1. Calculate and store $\phi(\boldsymbol{x}_1)$, ..., $\phi(\boldsymbol{x}_l)$ as the new input data.
2. Use $\boldsymbol{x}_1$, ..., $\boldsymbol{x}_l$ as the input data and calculate all $\phi(\boldsymbol{x}_1)$, ..., $\phi(\boldsymbol{x}_l)$ before training.
3. Use $\boldsymbol{x}_1$, ..., $\boldsymbol{x}_l$ as the input data and calculate $\phi(\boldsymbol{x}_i)$ in a training algorithm (e.g., decomposition method).

These methods have advantages/disadvantages. The first method does not require any modification of linear-SVM solvers, but needs a large $O(l\hat{n})$ disk/memory space to store $\phi(\boldsymbol{x}_i)$, $\forall i$. The second method also needs extra memory spaces, but avoids the long time for loading data from disk. The third method does not need extra memory, but requires some modifications of the decomposition implementation. That is, we need to calculate $\phi(\boldsymbol{x}_i)$ in (8). These three methods are useful under different circumstances. In Section 4.3, we experimentally show that for data with not too large n, the third way is the fastest. Although $\phi(\boldsymbol{x}_i)$ $\forall i$ can be stored in memory, accessing data from memory to cache and then

CPU may be slower than performing the calculation. However, for an application with very large n and very small $\bar{n}$, we demonstrate that the first or the second method may be more suitable. See the discussion later in this section.

While we may avoid storing $\phi(\boldsymbol{x}_i)$, one memory bottleneck remains. The vector $\boldsymbol{w}$ has a huge number of $O(n^d)$ components. If some features of $\phi(\boldsymbol{x}_i)$, $\forall i$ are zero, their corresponding elements in $\boldsymbol{w}$ are useless. Hence we can implement a sparse $\boldsymbol{w}$ to save the storage. In the following we analyze the possibility of having a sparse $\boldsymbol{w}$ by considering $d = 2$ and assuming that features of an instance have an equal opportunity to be zeros. The probability that $(\boldsymbol{x}_i)_r(\boldsymbol{x}_i)_s$ is zero for all $\phi(\boldsymbol{x}_i)$, $i = 1, \ldots, l$ is

$$\prod\nolimits_{i=1}^{l} \left(1 - \frac{n_i(n_i-1)}{n(n-1)}\right).$$

Note that $n_i(n_i - 1)/n(n - 1)$ is the probability that $(\boldsymbol{x}_i)_r(\boldsymbol{x}_i)_s$ is non-zero. Then the expected number of non-zero elements in $\boldsymbol{w}$ can be approximated by

$$C(n+2,2) - \frac{n(n-1)}{2} \prod\nolimits_{i=1}^{l} \left(1 - \frac{n_i(n_i-1)}{n(n-1)}\right), \tag{11}$$

where $n(n - 1)/2$ is the number of $x_r x_s$ terms in (4). This number is usually close to $C(n + 2, 2)$ due to the product of l values in (11). Moreover, this estimate is only accurate if features are independent. The assumption may hold for data sets with features from bitmaps or word frequencies, but is wrong for data with exclusive features (e.g., binary representation of a nominal feature). For data with known structures of features, in Section 4.2 we use real examples to illustrate how to more accurately estimate the number of $\boldsymbol{w}$'s non-zero elements.

In Section 5, we present an example of using a sparse $\boldsymbol{w}$. The dimensionality of the input data is $n = 46,155$. If $d = 2$, then storing $\boldsymbol{w}$ as a dense vector takes almost 20 GBytes of memory (assuming double precision). This problem has a very small $n_i \approx 13.3, \forall i$. Many $x_r x_s$ terms are zero in all $\phi(\boldsymbol{x}_i), i = 1, \ldots, l$, so $\boldsymbol{w}$ is very sparse. However, a naive implementation can be very inefficient. Assume $\tilde{n}$ is the number of non-zero elements in $\boldsymbol{w}$, where for this example $\tilde{n} = 1,438,456$. Accessing an element in a sparse vector requires an expensive $O(\tilde{n})$ linear search. We can use a hash table to store $\boldsymbol{w}$, but experiments show that the access time of $\boldsymbol{w}$ is still high. If $\phi(\boldsymbol{x}_i)$, $\forall i$ can be stored in memory, we construct a hash mapping from (r, s) to $j \in \{1, \ldots, \tilde{n}\}$ and re-generate training data using feature index j. That is, we construct a condensed representation for $\phi(\boldsymbol{x}_i)$, $\forall i$ and train a linear SVM with a dense $\boldsymbol{w} \in R^{\tilde{n}}$. The training is much more efficient because we can easily access any element of $\boldsymbol{w}$. In prediction, for any testing instance $\boldsymbol{x}$, we use the same hash table to conduct the inner product $\boldsymbol{w}^T\phi(\boldsymbol{x})$. This strategy corresponds to the first/second methods in the above discussion of handling $\phi(\boldsymbol{x}_i)$ $\forall i$.

The above technique to condense $\phi(\boldsymbol{x}_i)$ has been used in recent works on hash kernels (e.g., Shi et al., 2009). They differ from us in several aspects. First, their condensed representation is an approximation to $\phi(\boldsymbol{x}_i)$. Second, with an online setting, they may not accurately solve the problem (1).

3.5 Relations with the RBF kernel

RBF kernel (Gaussian kernel) may be the most used kernel in training nonlinear SVM. It takes the following form:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{\frac{-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{2\sigma^2}}.$$

Keerthi and Lin (2003) show that, as $\sigma^2 \to \infty$, SVM with the RBF kernel and the penalty parameter C approaches linear SVM with the penalty parameter $C/(2\sigma^2)$. This result implies that with a suitable parameter selection, the testing accuracy of using the RBF kernel is at least as good as using the linear kernel.

For polynomial kernels, Lippert and Rifkin (2006) discuss the relation with RBF. They consider the penalty parameter $(1/(2\sigma^2))^{-2d}C$ and check the situation as $\sigma^2 \to \infty$. For a positive integer d, the limit of SVM with the RBF kernel approaches SVM with a degree-d polynomial mapping of data. The polynomial mapping is related only to the degree d. This result seems to indicate that RBF is again at least as good as polynomial kernels. However, the polynomial mapping for (3) is more general due to two additional parameters γ and r. Thus the situation is unclear if parameter selections have been applied to both kernels. In Section 4, we give a detailed comparison between degree-2 polynomial mappings and RBF.

3.6 Parameter Selection

The polynomial kernel defined in (3) has three parameters (d, γ, and r). Now we intend to use low-degree mappings so d should be 2 or 3. Selecting the two remaining parameters is still complicated. Fortunately, we show in Appendix A that r can be fixed to one, so the number of parameters is the same as that of the RBF kernel. This result is obtained by proving that a polynomial kernel

$$\bar{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\bar{\gamma}\boldsymbol{x}_i^T\boldsymbol{x}_j + r)^d \text{ with parameters } (\bar{C}, \bar{\gamma}, r) \tag{12}$$

results in the same model as the polynomial kernel

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma\boldsymbol{x}_i^T\boldsymbol{x}_j + 1)^d \text{ with parameters } \gamma = \frac{\bar{\gamma}}{r} \text{ and } C = r^d\bar{C}. \tag{13}$$

3.7 Prediction

Assume a degree-d polynomial mapping is considered and #SV is the number of support vectors. For any testing data $\boldsymbol{x}$, the prediction time with/without kernels is

$$\begin{aligned} \textstyle\sum_{i:\alpha_i>0} \alpha_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) \quad &\Rightarrow \quad O(\#\text{SV} \cdot \bar{n}), && (14)\\ \boldsymbol{w}^T\phi(\boldsymbol{x}) \quad &\Rightarrow \quad O(\hat{n}), && (15)\end{aligned}$$

where $\bar{n}$ and $\hat{n}$ are respectively the average number of non-zero elements in $\boldsymbol{x}$ and $\phi(\boldsymbol{x})$. If $\hat{n} \leq \#\text{SV} \cdot \bar{n}$, then (15) is more efficient than (14). Several NLP studies (e.g., Isozaki and Kazawa, 2002) have used (15) for efficient testing.

Data set	n	$\bar{n}$	l	# testing
a9a	123	13.9	32,561	16,281
real-sim	20,958	51.5	57,848	14,461
news20	1,355,181	455.5	15,997	3,999
ijcnn1	22	13.0	49,990	91,701
MNIST38	752	168.2	11,982	1,984
covtype	54	11.9	464,810	116,202
webspam	254	85.1	280,000	70,000

Table 1: Summary of the data sets. n is the number of features, and $\bar{n}$ is the average number of non-zero features for each data instance. l is the number of data instances. The last column shows the number of testing data.

4. Experiments

In this section, we experimentally analyze the proposed approach for degree-2 polynomial mappings. We use two-class data sets, but in Section 5, we consider multi-class problems from an NLP application. We briefly discuss extensions to L1-regularized SVM in Section 4.5.

Except programs used in Section 5, all sources for experiments are available at http://www.csie.ntu.edu.tw/~cjlin/liblinear/exp.html.

4.1 Data Sets and Implementations

We select the following problems from LIBSVM tools[2] for experiments: a9a, real-sim, news20, ijcnn1, MNIST, covtype and webspam. The summary of data sets is in Table 1. Problems real-sim, news20, covtype and webspam have no original test sets, so we use a 80/20 split for training and testing. MNIST is a 10-class problem; we consider classes 3 and 8 to form a two-class data set MNIST38. While covtype is originally multi-class, we use a two-class version at LIBSVM tools.

We do not further scale these data sets as some of them have been pre-processed. Problems real-sim, news20 and webspam are document sets and each instance is normalized to unit length. We use a scaled version of covtype at LIBSVM tools, where each feature is linearly scaled to $[0, 1]$. The original MNIST data have all feature values in the range $[0, 255]$, but the version we download is scaled to $[0, 1]$ by dividing every value by 255.

We compare implicit mappings (kernel) and explicit mappings of data by LIBSVM (Chang and Lin, 2001) and an extension of LIBLINEAR (Fan et al., 2008), respectively. The two packages use similar stopping conditions, and we set the same stopping tolerance 0.1. Experiments are conducted on a 2.5G Xeon L5420 machine with 16G RAM using gcc compiler. Our experiments are run on a single CPU.

[2] LIBSVM tools can be found at http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

	Analysis of $\phi(\boldsymbol{x}_i)$			# non-zeros in $\boldsymbol{w}$			
Data set	$\bar{n}^2/2$	$\hat{n}$	$l\bar{n}$	$C(n{+}2,2)$	Estimated by		Real
					(11)	(16)	
a9a	96.2	118.1	4.52e+05	7.75e+03	7.75e+03	6.60e+03	5.56e+03
real-sim	1,325.1	2,923.6	2.98e+06	2.20e+08	1.15e+08		3.01e+07
news20	103,750.6	327,051.6	7.29e+06	9.18e+11	5.20e+09		3.13e+09
ijcnn1	84.5	105.0	6.50e+05	2.76e+02	2.76e+02	2.31e+02	2.31e+02
MNIST38	14,147.2	14,965.1	2.02e+06	2.84e+05	2.84e+05		1.54e+05
covtype	71.3	90.3	5.55e+06	1.54e+03	1.54e+03	7.54e+02	6.69e+02
webspam	3,623.5	3,836.4	2.38e+07	3.26e+04	3.26e+04		9.44e+03

Table 2: Analysis of $\phi(\boldsymbol{x}_i)$, $i = 1, \ldots, l$ and number of non-zero elements in $\boldsymbol{w}$.

4.2 Analysis of $\phi(\boldsymbol{x})$ and $\boldsymbol{w}$

Following the discussion in Section 3.2, we check $l\bar{n}$ and $\hat{n}$ to see if using the explicit mapping of data may be helpful. Table 2 presents these two values for the degree-2 polynomial mapping. We also present $\bar{n}^2/2$ as from (10) it can be a rough estimate of $\hat{n}$.

From Table 2, except document data real-sim and news20, $\bar{n}^2/2$ is close to $\hat{n}$. The huge difference between $\hat{n}$ and $l\bar{n}$ indicates that using explicit mappings is potentially faster.

Next we investigate the number of non-zero elements in $\boldsymbol{w}$. Table 2 presents the dimensionality of $\boldsymbol{w}$, two estimated numbers of non-zero elements and the actual number. For most data, the first estimation by (11) pessimistically predicts that $\boldsymbol{w}$ is fully dense. Two exceptions are real-sim and news20, where (11) is quite accurate. These two sparse document sets seem to have independent features (word occurrence) so that the assumption for (11) holds. For data with known structures, we demonstrate that a better estimate than (11) can be achieved. The problem a9a contains 14 groups of features[3] and each group contains several exclusive binary features (e.g., age in various ranges). Within each group, $x_r x_s = 0$ if $r \neq s$, so an upper bound of $\boldsymbol{w}$'s number of non-zero elements is

$$C(n+2,2) - \sum_{\text{feature groups}} C(\#\text{features in each group}, 2). \tag{16}$$

We show in Table 2 that (16) is closer to the actual number. The situation for two other problems (ijcnn1 and covtype) is similar.

As storing news20's non-zero $\boldsymbol{w}$ elements requires more memory than our machine's capacity, we do not include this set in subsequent experiments.

4.3 Calculating or Storing $\phi(\boldsymbol{x}_i)$

In Section 3.4, we discuss three methods to handle $\phi(\boldsymbol{x}_i)$. Table 3 compares the first/second and the third methods. We select C and kernel parameters by a validation procedure and present the training time of using optimal parameters. For the first/second methods, we count CPU time after all $\phi(\boldsymbol{x}_i)$ have been loaded/generated. For the third, we show CPU

[3] See descriptions in the beginning of each file from `http://research.microsoft.com/en-us/um/people/jplatt/adult.zip`.

Data set	$\hat{n}/\bar{n}$	Storing $\phi(\boldsymbol{x}_i)$		Calculating $\phi(\boldsymbol{x}_i)$	
		L2 cache misses	Training time (s)	L2 cache misses	Training time (s)
a9a	8.51	5.62e+07	2.2	2.51e+06	1.6
real-sim	56.79	2.60e+09	63.3	1.84e+09	59.8
ijcnn1	8.08	3.62e+08	14.0	2.32e+07	10.7
MNIST38	88.97	9.08e+08	20.4	7.90e+06	8.6
covtype	7.56	1.55e+11	6,422.4	2.98e+10	5,211.9
webspam[4]	45.07	1.30e+11	4,219.3	3.20e+09	3,228.1

Table 3: A comparison between storing and calculating $\phi(\boldsymbol{x}_i)$. The column $\hat{n}/\bar{n}$ indicates the ratio between the memory consumption for storing $\phi(\boldsymbol{x}_i)$ and $\boldsymbol{x}_i$. Time is in seconds.

time after loading all $\boldsymbol{x}_i$, as this method calculates $\phi(\boldsymbol{x}_i)$, $\forall i$ in the middle of the training procedure.

Table 3 lists $\hat{n}/\bar{n}$ to show the ratio between two methods' memory consumption on storing $\phi(\boldsymbol{x}_i)$ and $\boldsymbol{x}_i$, $\forall i$. In the same table we present each method's number of L2 cache misses and training time. The number of L2 cache misses is obtained by the simulation tool cachegrind in valgrind.[5] The method by calculating $\phi(\boldsymbol{x}_i)$ is faster in all cases. Moreover, it has a smaller number of L2 cache misses. Data can be more easily located in the cache when $\phi(\boldsymbol{x}_i)$ is not stored. We consider the method of calculating $\phi(\boldsymbol{x}_i)$ for subsequent experiments in this section.

4.4 Accuracy and Time of Using Linear, Degree-2 Polynomial, and RBF

We compare training time, testing time, and testing accuracy of using three mappings: linear, degree-2 polynomial, and RBF. We use LIBLINEAR for linear, LIBSVM for RBF, and both for degree-2 polynomial. For each data set, we choose parameters C, γ (for polynomial), or σ^2 (for RBF) by a five-fold cross validation on a grid of points. The best parameters are then used to train the whole training set and obtain the testing accuracy. To reduce the training time, LIBSVM allocates some memory space, called kernel cache, to store recently used kernel elements. In contrast, LIBLINEAR does not require this space. All it needs is to store $\boldsymbol{w}$. In this work, we run LIBSVM using 1 GBytes of kernel cache.

Using linear and RBF mappings, Table 4 presents the training time, testing accuracy, and the correspondent parameters. Linear and RBF have similar testing accuracy on data sets a9a and real-sim. The set real-sim contains document data with many features. Linear classifiers have been observed to perform well on such data with much less training time. For other data sets, the testing accuracy of using linear is clearly inferior to that of using RBF. Degree-2 polynomial mappings may be useful for these data. We can possibly improve the accuracy over linear while achieving faster training time than RBF.

[4] For webspam, as $\phi(\boldsymbol{x}_i)$ $\forall i$ require more memory than what our machine has, we use single precision floating-point numbers to store the data. All other experiments in this work use double precision.

[5] cachegrind can be found at `http://valgrind.org/`.

	Linear (LIBLINEAR)			RBF (LIBSVM)			
Data set	C	Time (s)	Accuracy	C	σ	Time (s)	Accuracy
a9a	32	5.4	84.98	8	0.03125	98.9	85.03
real-sim	1	0.3	97.51	8	0.5	973.7	97.90
ijcnn1	32	1.6	92.21	32	2	26.9	98.69
MNIST38	0.03125	0.1	96.82	2	0.03125	37.6	99.70
covtype	0.0625	1.4	76.35	32	32	54,968.1	96.08
webspam	32	25.5	93.15	8	32	15,571.1	99.20

Table 4: Comparison of linear SVM and nonlinear SVM with RBF kernel. Time is in seconds.

Data set	Degree-2 Polynomial					Accuracy diff.	
	C	γ	Training time (s)		Accuracy	Linear	RBF
			LIBLINEAR	LIBSVM			
a9a	8	0.03125	1.6	89.8	85.06	0.07	0.02
real-sim	0.03125	8	59.8	1,220.5	98.00	0.49	0.10
ijcnn1	0.125	32	10.7	64.2	97.84	5.63	−0.85
MNIST38	2	0.3125	8.6	18.4	99.29	2.47	−0.40
covtype	2	8	5,211.9	NA	80.09	3.74	−15.98
webspam	8	8	3,228.1	NA	98.44	5.29	−0.76

Table 5: Training time (in seconds) and testing accuracy of using the degree-2 polynomial mapping. The last two columns show the accuracy difference to results using linear and RBF. NA indicates that programs do not terminate after 300,000 seconds.

We then explore the performance of the degree-2 polynomial mapping. The first part of Table 5 shows the training time, testing accuracy, and optimal parameters using LIBLINEAR. As a comparison, we run LIBSVM with the same parameters and report training time.[6] Table 5 also presents the testing accuracy difference between degree-2 polynomial and linear/RBF. It is observed that for nearly all problems, the performance of the degree-2 polynomial mapping can compete with RBF, while for covtype, the performance is only similar to the linear mapping. Apparently, a degree-2 mapping does not give rich enough information to separate data in covtype.

Regarding the training time, LIBLINEAR with degree-2 polynomial mappings is faster than LIBSVM with RBF. Therefore, the proposed method may achieve fast training, while preserving some benefits of nonlinear mappings. Next, we compare the training time between LIBLINEAR and LIBSVM when the same degree-2 polynomial mapping is used. From Table 5, LIBLINEAR is much faster than LIBSVM. Thus for applications needing to use low-degree polynomial kernels, the training time can be significantly reduced.

[6] LIBSVM solves SVM with bias b, but LIBLINEAR solves (1). As the difference is minor, we run LIBSVM with the same parameters for LIBLINEAR. Moreover, the testing accuracy of LIBSVM is almost the same as LIBLINEAR.

Data set	LIBLINEAR		LIBSVM	
	linear	degree-2	degree-2	RBF
a9a	0.00	0.01	19.28	32.42
real-sim	0.02	1.13	107.67	84.52
ijcnn1	0.02	0.07	14.07	20.38
MNIST38	0.00	0.12	2.41	5.76
covtype	0.03	0.09	NA	998.68
webspam	0.05	1.14	NA	846.77

Table 6: Testing time (in seconds) using decomposition methods for linear and nonlinear SVM. Parameters in Tables 4 and 5 are used to build SVM models for prediction. NA: SVM models are not available due to lengthy training time (see Table 5).

We present testing time in Table 6. The explicit mapping approach is much faster as it calculates only $\boldsymbol{w}^T\boldsymbol{x}$ or $\boldsymbol{w}^T\phi(\boldsymbol{x})$.

4.5 L1-regularized SVM with Linear and Degree-2 Polynomial Mappings

Recently, L1-regularized SVM has gained attention because it can produce a sparse model (see, for example, the survey by Yuan et al., 2009, and references therein). An L1-regularized SVM[7] solves

$$\min_{\boldsymbol{w}} \quad \|\boldsymbol{w}\|_1 + C\sum\nolimits_{i=1}^{l} \max(1 - y_i\boldsymbol{w}^T\phi(\boldsymbol{x}_i), 0)^2, \tag{17}$$

where $\|\cdot\|_1$ denotes the 1-norm. As discussed in Section 3.4, after a degree-2 polynomial mapping the number of features may be very large. A sparse $\boldsymbol{w}$ reduces the memory consumption. In this section, we conduct a preliminary investigation on training degree-2 polynomial mappings via (17).

Due to the non-differentiable term $\|\boldsymbol{w}\|_1$, optimization techniques for (17) are different from those for L2-regularized SVM. If we pre-compute $\phi(\boldsymbol{x}_i)$, $\forall i$ (i.e., methods 1 and 2 in Section 3.4 for handling $\phi(\boldsymbol{x}_i)$), then any optimization technique for (17) can be directly applied. Recall that Section 4.3 shows that method 3 (calculating $\phi(\boldsymbol{x}_i)$ in the training algorithm) is faster if n is not large. We show an interesting example where this method significantly increases the number of operations. In Yuan et al. (2009), a primal decomposition (coordinate descent) method is considered the fastest for solving (17). It updates one element of $\boldsymbol{w}$ at a time. If $\boldsymbol{w}$ is the current solution and the jth element is selected, the following one-variable problem is minimized:

$$\min_{d} \quad |w_j + d| + C\sum\nolimits_{i=1}^{l} \max\left(1 - y_i\boldsymbol{w}^T\phi(\boldsymbol{x}_i) - y_i d\phi(\boldsymbol{x}_i)_j, 0\right)^2.$$

Assume $\phi(\boldsymbol{x})_j$ involves $x_r x_s$. To obtain all $\phi(\boldsymbol{x}_i)_j$, $\forall i$, we must go through non-zero elements of the rth and the sth features of the original data. This operation costs $O(\bar{l})$,

[7] We consider L2-loss in (17) by following Yuan et al. (2009).

	Linear			Degree-2 polynomial				
Data set	Time	Sparsity	Accuracy	Time (s): $\phi(\boldsymbol{x}_i)$ is		Sparsity	Accuracy	L2 SVM
	(s)	(%)	(%)	stored	calculated	(%)	(%)	Sparsity
a9a	0.73	83.74	85.00	3.94	19.33	10.43	85.10	71.74
real-sim	1.06	25.16	97.04	524.54	2,288.27	0.01	97.43	26.17
ijcnn1	0.86	100.00	91.79	9.17	18.09	83.70	97.59	83.70
MNIST38	0.86	55.85	96.93	38.11	81.10	0.23	99.50	54.23
covtype	60.07	98.15	75.66	70.13	2,196.08	39.22	79.73	43.44
webspam	47.08	38.98	92.55	772.92	1,296.40	12.60	98.32	28.96

Table 7: L1-regularized SVM: a comparison between linear and degree-2 polynomial mappings. Time is in seconds. Sparsity is the percentage of non-zero elements in $\boldsymbol{w}$. For the degree-2 polynomial mapping, we show the training time of both calculating and storing $\phi(\boldsymbol{x})$.

where $\bar{l}$ is the average number of non-zero elements per feature of $\boldsymbol{x}_i$, $\forall i$. However, the expected number of non-zero elements of $\phi(\boldsymbol{x}_i)_j$, $\forall i$ is only

$$\left(\bar{l}/l\right)^2 \cdot l = \bar{l}^2/l.$$

If data are sparse, $\bar{l}^2/l$ is much smaller than $\bar{l}$. Therefore, the cost by using methods 1 and 2 to pre-compute $\phi(\boldsymbol{x}_i)$, $\forall i$ is less than method 3. This is mainly because the primal coordinate descent approach accesses data in a feature-based way. The sparse patterns of two features are needed. In contrast, decomposition methods used earlier for solving the dual problem of L2-regularized SVM is instance-based. To obtain $x_r x_s$, one needs only the sparse pattern of an instance $\boldsymbol{x}$. Some optimization approaches discussed in Yuan et al. (2009) for (17) are instance-based. An interesting future study would be to investigate their performances.

We extend a primal decomposition implementation for (17) in LIBLINEAR to handle degree-2 polynomial mappings. We use default settings (e.g., stopping tolerance) in LIBLINEAR. Table 7 compares linear and degree-2 polynomial mappings by showing training time, $\boldsymbol{w}$'s sparsity, and testing accuracy. For the training time of using degree-2 polynomials, we present results by storing and calculating $\phi(\boldsymbol{x}_i)$. Clearly, calculating $\phi(\boldsymbol{x}_i)$ is much slower, a result consistent with our analysis. The training time for real-sim is much longer than that in Table 5 (L2-regularized SVM). This result is due to the huge number of variables in solving the primal problem (17). There are some tricks to improve the training speed for this problem though we do not get into details. For sparsity, we also show the result using L2-regularized SVM as a comparison.[8] L1-regularized SVM gives excellent sparsity for some problems. For example, MNIST38 has $n = 752$ features. By solving (17) with linear mapping, 420 features remain. If using a degree-2 polynomial mapping, the dimensionality is $C(n+2, 2) = 283,881$. Solving an L2-regularized SVM gives a $\boldsymbol{w}$ with 153,564 non-zero elements, but using L1 regularization $\boldsymbol{w}$ has only a very small number of

[8] The sparsity of L2-regularized SVM is in fact the column of "real" numbers of non-zero elements in Table 2 divided by the dimensionality $C(n+2, 2)$.

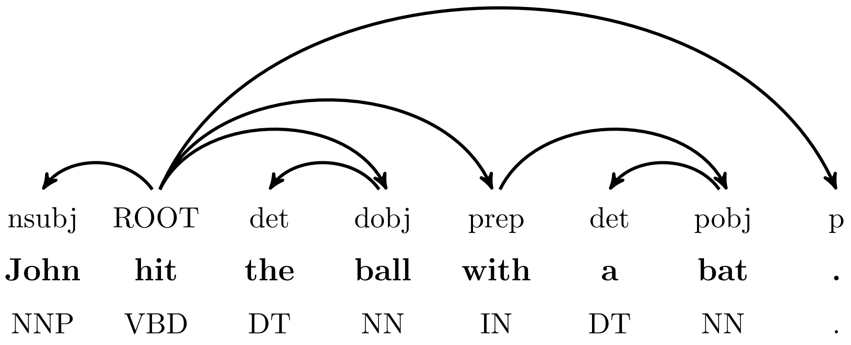

Figure 1: A dependency graph with arc labels and part-of-speech tags for a sentence.

650 non-zero elements. Finally, for testing accuracy, results are similar to (or slightly lower than) those in Tables 4-5.

Due to the nice sparsity results, L1 regularization for low-degree polynomial mappings may be a promising future direction.

5. An NLP Application: Data-driven Dependency Parsing

In this section we study a real-world natural language processing (NLP) task on dependency parsing. Given a sentence, a dependency graph represents each word and its syntactic modifiers through labeled directed edges. Figure 1 shows an example. Data-driven dependency parsing is a common method to construct dependency graphs. Different from grammar-based parsing, it learns to produce the dependency graph solely by using the training data. Data-driven dependency parsing has become popular because it is chosen as the shared task at CONLL-X[9] and CONLL2007.[10] More information about dependency parsing can be found in, for example, McDonald and Nivre (2007).

Dependency parsing appears in many online NLP applications. In such cases testing (parsing) speed is very important. We will see that this requirement for testing speed makes our approach very useful for this application.

5.1 A Multi-class Problem

We study a transition-based parsing method proposed by Nivre (2003). The parsing algorithm builds a labeled dependency graph in one left-to-right pass over the input with a stack to store partially processed tokens. At each step, we need to decide which transition to perform. As in Nivre et al. (2007), we use the following transitions:

- SHIFT: Pushes the next input token to the top of the stack and advances to the next input token.

- REDUCE: Pops the top element from the stack.

- LEFT-ARC(r): Adds an arc with label r from the next input token to the token on top of the stack and pops the top element off the stack.

[9] CONLL-X can be found at `http://nextens.uvt.nl/~conll/`.

[10] CONLL2007 can be found at `http://nextens.uvt.nl/depparse-wiki/SharedTaskWebsite`.

input(1).tag
input(2).tag
input(3).tag
input.word
input(1).word
stack.tag
stack(1).tag
stack.label
stack.word
stack.head.word
stack.leftmost-child.label
stack.rightmost-child.label
input.leftmost-child.label

Table 8: Feature types used by the dependency parser.

- RIGHT-ARC(r): Adds an arc with label r from the top token on the stack to the next input token. Then pushes the current input token to the stack and advances to the next input token.

The parser decides the next transition by extracting features from the current parse state. The parse state consists of the stack, the remaining input tokens, and the partially built dependency graph. We use the standard method for converting symbolic features into numerical features by binarization. For each feature type F (see Table 8), that has value v in the current state, we generate a feature predicate, $F = v$. These feature predicates are used as binary features in the classifier. It is this expansion of feature types to binary feature predicates that leads to the large number of features in the classifiers. Especially the lexicalized (i.e., word-based) features generate large numbers of sparse features.

Thus the core of the parser is a multi-class classification problem, which maps features to transitions. Nivre et al. (2006) use LIBSVM with a degree-2 polynomial kernel to train the multi-class classification problems, and get good results at CONLL-X.

In this experiment, we use data from the English Penn Treebank (Marcus et al., 1993). The treebank is converted to dependency format using Penn2Malt,[11] and the data is split into sections 02–21 for training and section 23 for testing. During training we construct a canonical transition sequence from the dependency graph of each sentence in the training corpus, adopting an arc-eager approach for disambiguation. For each transition, we extract the features in Table 8 from the current parse state, and use this for training the classifiers.

When parsing a sentence, the classifiers are used for predicting the next transition, based on the features extracted from the current parse state. When all the input tokens have been processed the dependency graph is extracted from the transition sequence.

In order to reduce training time the data is split into multiple sets. For example, if a feature j takes two values a and b, we can divide the training data into $\{\boldsymbol{x} \mid x_j = a\}$ and $\{\boldsymbol{x} \mid x_j = b\}$, and get two models M^a and M^b. Then in the prediction phase, we decide to use M^a or M^b according to x_j of the testing instance. Yamada and Matsumoto (2003) mention that applying this method reduces the training time without a significant loss in accuracy. We divide the training data into 125 smaller training sets according to the part-of-speech tag of the current input token. Also, the label for RIGHT-ARC and LEFT-ARC transitions is predicted separately from the transition. The number of classes ranges from 2 to 12. Table 9 lists the statistics of the largest multi-class problem among 125.

[11] See `http://w3.msi.vxu.se/~nivre/research/Penn2Malt.html`

n	$\bar{n}$	l	#nz
46,155	13.3	294,582	3,913,845

Table 9: Summary of the dependency parsing data set. We show statistics of the largest problem among the 125 sets divided from the training data. The column #nz ($= l\bar{n}$) indicates the total number of non-zero feature values in the training set.

5.2 Implementations

We consider the degree-2 polynomial mapping in (5). Since the original $\boldsymbol{x}_i, \forall i$ have 0/1 feature values, we use (5) instead of (4) to preserve this property.[12] In our implementation, by extending LIBLINEAR, 0/1 values are still stored as double-precision numbers. However, for large data sets, we can save memory by storing only non-zero indices. Due to using (5), the only parameter is C.

The dimensionality of using the degree-2 polynomial mapping is huge. We discussed in Section 3.4 that 20 GBytes memory is needed to store a dense $\boldsymbol{w}$. Assume the "one-against-the rest" approach is applied for multi-class classification. We need 125 × (# classes) vectors of $\boldsymbol{w}$ in the prediction (parsing) stage as training data are separated into 125 sets. Obviously we do not have enough memory for them. As the data are very sparse, the actual number of non-zero elements in $\boldsymbol{w}$ is merely 1,438,456 (considering the largest of the 125 training sets). Only these non-zero features in $\phi(\boldsymbol{x})$ and $\boldsymbol{w}$ are needed in training/testing, so the memory issue is solved. In the practical implementation, we construct a hash table to collect non-zero features of $\phi(\boldsymbol{x}_i), i = 1, \ldots, l$ as a new set for training.[13] In prediction, we use the same hash map to calculate $\boldsymbol{w}^T\phi(\boldsymbol{x})$. This implementation corresponds to the first/second methods discussed in Section 3.4. See more details in the end of Section 3.4.

Other settings (e.g., stopping tolerance and LIBSVM's kernel cache) are the same as those in Section 4. LIBSVM uses the "one-against-one" approach for training multi-class problems, while LIBLINEAR uses "one-against-the rest." We use a dependency parsing system at Google, which calls LIBSVM/LIBLINEAR for training/testing. Parts of this experiment were performed when some authors worked at Google.

As the whole parsing system is quite complex, we have not conducted a complete parameter optimization. Instead, we have roughly tuned each parameter to produce good results.

5.3 Experiments

We compare two approaches. The first uses kernels, while the second does not.

- LIBSVM: RBF and degree-2 polynomial kernels.
- LIBLINEAR: linear mapping (i.e., the original input data) and the degree-2 polynomial mapping via (5).

[12] In fact, if using (4), we can still use some ways so that $\sqrt{2}$ is not stored. However, the implementation is more complicated.

[13] For a fair comparison, the reported training time includes time for this pre-processing stage.

	LIBSVM		LIBLINEAR	
	RBF	Poly ($d = 2$)	Linear	Poly: Eq. (5)
Parameters	$C = 0.5$ $1/(2\sigma^2) = 0.18$	$C = 0.5$ $\gamma = 0.18$ $r = 0.3$	$C = 0.5$	$C = 0.05$
Training time	3h34m53s	3h21m51s	3m36s	3m43s
Parsing speed	0.7x	1x	1652x	103x
UAS	89.92	91.67	89.11	91.71
LAS	88.55	90.60	88.07	90.71

Table 10: Accuracy, training time, and parsing speed (relative to LIBSVM with polynomial kernel) for the dependency parsing.

Table 10 lists parameters for various kernels, training/testing time, and testing accuracy. Training and testing are done using gold standard part-of-speech tags, and only non-punctuation tokens are used for scoring. The accuracy of dependency parsing is measured by two evaluation metrics:

1. Labeled attachment score (LAS): the percentage of tokens with correct dependency head and dependency label.

2. Unlabeled attachment score (UAS): the percentage of tokens with correct dependency head.

For LIBSVM the polynomial kernel gives better accuracy than the RBF kernel, consistent with previous observations, that polynomial mappings are important for parsing (Kudo and Matsumoto, 2000; McDonald and Pereira, 2006; Yamada and Matsumoto, 2003; Goldberg and Elhadad, 2008). Moreover, LIBSVM using degree-2 polynomial kernel produces better results in terms of UAS/LAS than LIBLINEAR using just a linear mapping of features. However, parsing using LIBSVM is slow compared to LIBLINEAR. We can speed up parsing by a factor of 1,652 with only a 2.5% drop in accuracy. With a degree-2 polynomial mapping (5), we achieve UAS/LAS results similar to LIBSVM, while still maintaining high parsing speed, 103 times faster than LIBSVM.

From Table 10, training LIBLINEAR is a lot faster than LIBSVM. This large reduction in training time allows us to easily conduct experiments and improve the settings. Some may criticize that the comparison on training time is not fair as LIBSVM uses "one-against-one" for multi-class classification, while LIBLINEAR uses "one-against-the rest." It is known (e.g., Hsu and Lin, 2002) that for nonlinear SVM, LIBSVM with "one-against-one" is faster than "one-against-the rest." Thus even if we modify LIBSVM to perform "one-against-the rest," its training is still much slower than LIBLINEAR.

5.4 Related Work

Earlier works have improved the testing speed of SVM with low-degree polynomial kernels. Most of them target natural language processing (NLP) applications. Isozaki and Kazawa

(2002) propose an approach similar to obtaining $\boldsymbol{w}$ by the expression in (8).[14] A direct implementation of their method requires a memory space as large as the dimensionality of $\boldsymbol{w}$, but we cannot afford such a space for our application. Kudo and Matsumoto (2003) consider the expression of $\boldsymbol{w}$ in (8) and propose an approximate prediction scheme using only a sub-vector of $\boldsymbol{w}$. Their method is useful for data with 0/1 features. Goldberg and Elhadad (2008) propose speeding up the calculation of low-degree polynomial kernels by separating features to rare and common ones. Goldberg and Elhadad's approach is motivated by some observations of NLP data. It avoids the memory problem, but the effectiveness on general data is not clear yet.

The above existing works focus on improving testing speed. They suffer from the slow training of using traditional SVM solvers. For example, Kudo and Matsumoto (2000) mention that "the experiments ... have actually taken long training time," so they must select a subset using properties of dependency parsing. Our approach considers linear SVM on explicitly mapped data, applies state of the art training techniques, and can simultaneously achieve fast training and testing.

6. Discussions and Conclusions

Past research has shown that SVM using linear and highly nonlinear mappings of data has the following properties:

Linear	Highly nonlinear
Fast training/testing	Slow training/testing via kernels
Low accuracy	High accuracy

Many have attempted to develop techniques in the between. Most start from the nonlinear side. They propose methods to manipulate the kernels (e.g., Lee and Mangasarian, 2001; Keerthi et al., 2006). In contrast, ours is from the linear side. The strategy is simple and requires only minor modifications of existing packages for linear SVM.

This work focuses on the degree-2 polynomial mapping. An interesting future study is the efficient implementation for degree-3 mappings. Considering other mapping functions to expand data vectors could be investigated as well. As kernels are not used, we might have a greater flexibility to design the mapping function.

Table 2 shows that storing $\boldsymbol{w}$ may require a huge amount of memory. For online training, some (e.g., Langford et al., 2009) have designed feature hashing techniques to control the memory use of $\boldsymbol{w}$. Recently, feature hashing has been popular for projecting a high dimensional feature vector to a lower dimensional one (e.g., Weinberger et al., 2009; Shi et al., 2009). For certain sequence data, one can consider n-gram (i.e., n consecutive features) instead of general polynomial mappings. Then not only the number of features becomes smaller, but also controlling $\boldsymbol{w}$'s sparsity is easier. Existing experiments on document data can be found in, for example, Ifrim et al. (2008) and Shi et al. (2009).

[14] They do not really form $\boldsymbol{w}$, but their result by expanding the degree-2 polynomial kernel leads to something very similar.

We successfully apply the proposed procedure to an NLP application. It has certain requirements on the training and testing speed, but we also hope to achieve better testing accuracy. The proposed procedure is very useful for applications of this type.

Acknowledgments

The authors thank Aditya Menon, Ming-Wei Chang, anonymous reviewers, and associate editor for helpful comments. They also thank Naomi Bilodeau for proofreading the paper. This work was supported in part by the National Science Council of Taiwan via the grant 98-2221-E-002-136-MY3.

Appendix A. Connection Between (12) and (13)

We prove the result by showing that the dual optimization problems of using (12) and (13) are the same. Since

$$(\bar{\gamma}\boldsymbol{x}_i^T\boldsymbol{x}_j + r)^d = r^d\left(\frac{\bar{\gamma}}{r}\boldsymbol{x}_i^T\boldsymbol{x}_j + 1\right)^d,$$

we have

$$\bar{Q}_{ij} = y_i y_j \bar{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = r^d Q_{ij}.$$

The dual optimization problem of using $\bar{Q}$ can be written as

$$\begin{aligned} \min_{\bar{\boldsymbol{\alpha}}} \quad & \frac{1}{r^d}\left(\frac{1}{2}(r^d\bar{\boldsymbol{\alpha}})^T Q(r^d\bar{\boldsymbol{\alpha}}) - \boldsymbol{e}^T r^d\bar{\boldsymbol{\alpha}}\right) \\ \text{subject to} \quad & 0 \le r^d\bar{\alpha}_i \le r^d\bar{C}, \quad i = 1\ldots, l. \end{aligned}$$

Using $\boldsymbol{\alpha} = r^d\bar{\boldsymbol{\alpha}}$ and $C = r^d\bar{C}$, this problem becomes the dual problem when using Q.

References

Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

Leon Bottou. Stochastic gradient descent examples, 2007. http://leon.bottou.org/projects/sgd.

Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Corina Cortes and Vladimir Vapnik. Support-vector network. *Machine Learning*, 20:273–297, 1995.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9: 1871–1874, 2008. URL http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf.

Michael Ferris and Todd Munson. Interior point methods for massive support vector machines. *SIAM Journal on Optimization*, 13(3):783–804, 2003.

E. Michael Gertz and Joshua D. Griffin. Support vector machine classifiers for large data sets. Technical Report ANL/MCS-TM-289, Argonne National Laboratory, 2005.

Yoav Goldberg and Michael Elhadad. splitSVM: Fast, space-efficient, non-heuristic, polynomial kernel computation for NLP applications. In *Proceedings of the 46st Annual Meeting of the Association of Computational Linguistics (ACL)*, 2008.

Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and Sellamanickam Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *Proceedings of the Twenty Fifth International Conference on Machine Learning (ICML)*, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf`.

Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, 2002.

Georgiana Ifrim, Gökhan Bakır, and Gerhard Weikum. Fast logistic regression for text categorization with variable-length n-grams. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 354–362, 2008.

Hideki Isozaki and Hideto Kazawa. Efficient support vector classifiers for named entity recognition. In *Proceedings of COLING*, pages 390–396, 2002.

Thorsten Joachims. Making large-scale SVM learning practical. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, Cambridge, MA, 1998. MIT Press.

Thorsten Joachims. Training linear SVMs in linear time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006.

Jin Hyuk Jung, Dianne P. O'Leary, and André L. Tits. Adaptive constraint reduction for training support vector machines. *Electronic Transactions on Numerical Analysis*, 31: 156–177, 2008.

S. Sathiya Keerthi and Chih-Jen Lin. Asymptotic behaviors of support vector machines with Gaussian kernel. *Neural Computation*, 15(7):1667–1689, 2003.

S. Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Karuturi Radha Krishna Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13:637–649, 2001.

S. Sathiya Keerthi, Olivier Chapelle, and Dennis DeCoste. Building support vector machines with reduced classifier complexity. *Journal of Machine Learning Research*, 7:1493–1515, 2006.

Taku Kudo and Yuji Matsumoto. Japanese dependency structure analysis based on support vector machines. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora*, 2000.

Taku Kudo and Yuji Matsumoto. Fast methods for kernel-based text analysis. In *Proceedings of the 41st Annual Meeting of the Association of Computational Linguistics (ACL)*, 2003.

John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. *Journal of Machine Learning Research*, 10:771–801, 2009.

Yuh-Jye Lee and Olvi L. Mangasarian. RSVM: Reduced support vector machines. In *Proceedings of the First SIAM International Conference on Data Mining*, 2001.

Ross A. Lippert and Ryan M. Rifkin. Infinite-σ limits for Tikhonov regularization. *Journal of Machine Learning Research*, 7:855–876, 2006.

O. L. Mangasarian and D. R. Musicant. Successive overrelaxation for support vector machines. *IEEE Transactions on Neural Networks*, 10(5):1032–1037, 1999.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

Ryan McDonald and Joakim Nivre. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the Joint Conferences on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

Ryan McDonald and Fernando Pereira. Online learning of approximate dependency parsing algorithms. In *Proceedings of 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 81–88, 2006.

Yvonne Moh and Joachim M. Buhmann. Kernel expansion for online preference tracking. In *Proceedings of The International Society for Music Information Retrieval (ISMIR)*, pages 167–172, 2008.

Joakim Nivre. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, 2003.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Svetoslav Marinov. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225, 2006.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gulsen Eryigit, Sandra Kubler, Svetoslav Marinov, and Erwin Marsi. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: primal estimated sub-gradient solver for SVM. In *Proceedings of the Twenty Fourth International Conference on Machine Learning (ICML)*, 2007.

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and S.V.N. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10: 2615–2637, 2009.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the Twenty Sixth International Conference on Machine Learning (ICML)*, pages 1113–1120, 2009.

Hiroyasu Yamada and Yuji Matsumoto. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, 2003.

Guo-Xun Yuan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A comparison of optimization methods and software for large-scale l1-regularized linear classification. 2009. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/l1.pdf`. Under revision for Journal of Machine Learning Research.