Fast and accurate text classification via multiple linear discriminant projections

Soumen Chakrabarti* Shourya Roy Mahesh V. Soundalgekar
IIT Bombay

Abstract

Support vector machines (SVMs) have shown superb performance for text classification tasks. They are accurate, robust, and quick to apply to test instances. Their only potential drawback is their training time and memory requirement. For n training instances held in memory, the best-known SVM implementations take time proportional to n^a, where a is typically between 1.8 and 2.1. SVMs have been trained on data sets with several thousand instances, but Web directories today contain millions of instances which are valuable for mapping billions of Web pages into Yahoo!-like directories. We present SIMPL, a nearly linear-time classification algorithm which mimics the strengths of SVMs while avoiding the training bottleneck. It uses Fisher's linear discriminant, a classical tool from statistical pattern recognition, to project training instances to a carefully selected low-dimensional subspace before inducing a decision tree on the projected instances. SIMPL uses efficient sequential scans and sorts, and is comparable in speed and memory scalability to widely-used naive Bayes (NB) classifiers, but it beats NB accuracy decisively. It not only approaches and sometimes exceeds SVM accuracy, but also beats SVM running time by orders of magnitude. While developing SIMPL, we also make a detailed experimental analysis of the cache performance of SVMs.

1 Introduction

Text classification is a standard problem in information retrieval (IR). A *learner* is first presented with *training* documents d, each labeled as containing or not containing material relevant to a given topic; the label is denoted $c \in \{-1, 1\}$ (we can turn multi-topic problems into an ensemble of two-topic problems by building a yes/no classifier for each topic—this is standard). The learner processes the training documents, generally collecting term statistics and estimating various model parameters. Later, *test* instances are presented without the label, and the learner has to guess if each test document is or is not relevant to the given topic. Naive Bayes (NB), rule induction, decision trees, and support vector machines (SVMs) are some of the best-known classifiers employed to date.

Not surprisingly, the techniques that are easiest to code and fastest to execute are not very accurate, and vice versa. SVMs are the most accurate classifiers known for text applications [4, 8]: they beat NB accuracy by a decisive margin. The accuracy boost is intriguing, because both SVM and NB classifiers learn a hyperplane which separates the positive examples ($c = 1$) from the negative ones ($c = -1$), represented as vectors in a high-dimensional term space. The difference is that SVM picks a hyperplane embedded midway in the thickest possible slab passing between positive and negative examples and containing no training point.

NB takes time essentially linear in the number n of training documents [13, 14], whereas SVMs take time proportional to n^a, where a is typically between 1.8 and 2.1. Thanks to some clever implementations [17, 9], SVMs have been trained on several thousand instances despite their near-quadratic complexity. However, to achieve this, they hold the entire training data in main memory.

Scalability and memory footprint can become critical issues as enormous training sets become increasingly available. Web directories such as the Open Directory (also called Dmoz) and Yahoo! contain millions of training instances which occupy tens of gigabytes, whereas even high-end servers are mostly limited to 1–2 GB of RAM. Sampling down the training set hurts accuracy in such high-dimensional regimes: every additional training document helps, and most features reveal some useful class information [8]. In summary, despite

[*] Contact author, soumen@cse.iitb.ac.in

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

the theoretical elegance and superiority of SVMs, their IO behavior and CPU scaling are important concerns.

1.1 Our contribution

We design, implement, and evaluate a new, simple text classification algorithm which needs very little RAM, deals gracefully with out-of-RAM training data (which it accesses strictly linearly), beats NB accuracy decisively, and even matches SVM accuracy. Our main idea is to:

1. Find a series of projections of the training data by using Fisher's classical linear discriminant as a subroutine
2. Project all training instances to the low-dimensional subspace found in the previous step
3. Induce a decision tree on the projected low-dimensional data.

We call this general framework SIMPL (Simple Iterative Multiple Projection on Lines). SIMPL has several important features: it has very small footprint, linear in the number of terms (dimensions) m *plus* the number of documents n (with more careful coding, even the linear dependence on n can be removed); it makes only fast sequential scans over the input; its CPU time is linear in the total size of the training data, it can be expressed simply in terms of joins, sorts, and `GROUP BY` operations, and it can be parallelized easily.

To give a quick impression, SIMPL has been implemented using only 600 lines of C++ code, and trained on a 65524-document collection in 250 seconds, for which SVM took 3850 seconds. We undertake a careful comparison between SIMPL and SVM with regard to accuracy and performance. We find that in spite of its simplicity and efficiency, SIMPL is comparable (and sometimes superior) to SVM in terms of accuracy. It usually achieves such high accuracy with only two linear projections.

The ability to scale to training sets much larger than main memory is a key concern for the data mining community, which has resulted in excellent out-of-core implementations for traditional classifiers such as decision trees [20]. In the last few years, the machine learning and text mining communities have evolved other powerful classifiers, such as SVMs and maximum entropy learners. The scaling and IO behavior of the new and important class of SVM learners are not clearly understood. To this end, we carefully study the performance of SVM accessing documents from a LRU cache having limited size. If SVM is given a cache of size comparable to the RAM required by SIMPL, it spends a significant portion of its time servicing cache misses, and the performance gap between SIMPL and SVM grows further.

1.2 Related work

Although we are not aware of a hybrid learning strategy similar to our proposal, a few ideas that we discuss here were early hints that a projection-based approach could be promising. A 1988 theorem by Frankl and Maehara [5] showed that a projection of a set of n points in $\mathbb{R}^m$ to a *random* subspace of dimension about $(9/\epsilon^2)\log n$ preserves (to within a $1 \pm \epsilon$ factor) all relative inter-point distances with high probability. On a related note, Kleinberg projected these points to $\Theta(m \log^2 m)$ randomly directed lines to answer approximate nearest-neighbor queries efficiently [10].

For a classification task, we need not preserve *all* distances carefully. We simply need a subspace which separates the positive and negative instances well. In an early study by Schütze, Hull and Pedersen [19], even *single* linear discriminants compared favorably with neural networks for the document routing problem. Lewis and others [11] reported accurate prediction using a variety of regression strategies for good (single) linear predictors. The recent success of SVMs adds further evidence that very few projections could be adequate in the text domain.

In 1999, Shashua established that the decision surface found by a linear SVM is the *same* as the Fisher discriminant for only the "support vectors" (see §2.3) found by a SVM [21]. Although this result does not directly yield a better SVM algorithm, it gave us the basic intuition behind our idea. Our work is most closely related to linear discriminants [3] and SVMs, which we discuss in detail in §2 and §3. Independently, Cooke [2] has suggested discarding well-separated training points before finding Fisher's linear discriminant, but has not used multiple projections as a highly compressed representation for a more powerful learning algorithm such as a decision tree.

Pavlov, Mao and Dom also appreciate the importance of scaling up SVMs, but their approach is to run SVM as-is on 2–4% samples of the training set (which fit in memory) and then use boosting [16]. Their accuracy at best matches a classifier induced on the entire data set. Mangasarian and Musicant [12] propose an enhanced SVM formulation that learns on millions of data points in a few minutes, but they use several gigabytes of RAM. Fung and Mangasarian [6] propose to relax the SVM optimization problem to a simpler one involving linear optimization. Both these techniques depend on inverting either an $n \times n$ or an $m \times m$ matrix (n is the number of instances and m is the number of dimensions). In traditional data mining, $n \gg m$, and the latter inversion is preferred. For typical text classification benchmarks, both n and m range into tens of thousands, and matrix inversion takes time cubic in n or m. Fung, Mangasarian and Musicant experimented with m typically between 6 and 34 (maximum 123). Our data sets have 30000 to 1229663 dimensions.

Our approach may be regarded as a punch between oblique decision trees (ODTs) [15], which tries to find non-orthogonal hyperplane cuts in the decision-tree setting, and an extreme case of *boosting* [18], in which instances separated with the help of existing linear projections are completely removed from consideration. Inducing an ordinary decision tree over the raw term space of a large document collection is already extremely time-consuming. ODTs draw on an even more complex hypothesis space than decision trees (an arrangement of simplicial polytopes) and involve a regression over potentially all m dimensions at *each* node of the decision tree. Consequently, SIMPL is much faster than ODT induction. It is also somewhat faster to apply on test instances than ODTs because we only need to compute a small, fixed number of projections (usually 2). We also found SIMPL to be more accurate than decision trees. A comparison with ODTs may be worthwhile.

2 Preliminaries

2.1 Naive Bayes (NB) classifiers

Bayesian classifiers estimate a class-conditional document distribution $\Pr(d|c)$ from the training documents and use Bayes rule to estimate $\Pr(c|d)$ for test documents. The documents are modeled using their terms. The multinomial naive Bayes model assumes that a document is a bag or multiset of terms, and the term counts are generated from a multinomial distribution after fixing the document length ℓ_d, which, being fixed for a given document, lets us write

$$\Pr(d|c, \ell_d) \quad = \quad \binom{\ell_d}{\{n(d,t)\}} \prod_{t \in d} \theta_{c,t}^{n(d,t)}, \qquad (1)$$

where $n(d,t)$ is the number of times t occurs in d, and $\theta_{c,t}$ are suitably estimated [1, 14] multinomial probability parameters with $\sum_t \theta_{c,t} = 1$ for all c. For the two-class scenario throughout this paper, we only need to compare $\Pr(c = -1|d)$ against $\Pr(c = 1|d)$, or equivalently, $\log \Pr(c = -1|d)$ against $\log \Pr(c = 1|d)$, which simplifies to a comparison between

$$\begin{array}{l} \log \Pr(c=1) + \sum_{t \in d} n(d,t) \log \theta_{1,t} \quad \text{and} \\ \log \Pr(c=-1) + \sum_{t \in d} n(d,t) \log \theta_{-1,t}, \end{array} \qquad (2)$$

where $\Pr(c = \ldots)$, called the class *priors*, are the fractions of training instances in the respective classes. Simplifying (2), we see that NB is a linear classifier: it makes a decision between $c = 1$ and $c = -1$ by thresholding the value of $\alpha_{\text{NB}} \cdot d + b$ for a suitable vector α_{NB} (which depends on the parameters $\theta_{c,t}$) and constant b. Here d is overloaded to denote a vector of term frequencies (see §4) and '$\cdot$' denotes a dot-product.

2.2 Regression techniques

We can regard the classification problem as inducing a linear regression from d to c of the form $c = \alpha \cdot d + b$, where α and b are estimated from the data $\{(d_i, c_i), i = 1, \ldots, n\}$. This view has been common in a variety of IR applications. A common objective is to minimize the square error between the observed and predicted class variable: $\sum_d (\alpha \cdot d + b - c)^2$. The least-square optimization frequently uses gradient-descent methods, such as the Widrow-Hoff (WH) update rule. The WH approach starts with some rough estimate $\alpha^{(0)}$, considers (d_i, c_i) one by one and updates $\alpha^{(i-1)}$ to $\alpha^{(i)}$ as follows:

$$\alpha^{(i)} \quad = \quad \alpha^{(i-1)} + 2\eta(\alpha^{(i-1)} \cdot d_i - c_i) d_i. \qquad (3)$$

The final α used for classification is usually the average of all αs found along the way. Schütze, Lewis and others [19, 11] have applied WH and other update methods (such as the Exponentiated Gradient method) to design high-accuracy linear classifiers for text. We will follow the WH approach, but we will not minimize the square error, because we are not dependent on a single linear predictor. Instead, our goal is to maximize separation between the classes in the projected subspace, for which we will optimize Fisher's linear discriminant.

2.3 Linear support vector machines

Like NB, linear SVMs also make a decision by thresholding $\alpha_{\text{SVM}} \cdot d + b$ (the estimated class is $+1$ or -1 according as the quantity is greater or less than 0) for a suitable vector α_{SVM} and constant b. α_{SVM} is chosen far more carefully than NB. Initially, let us assume that the n training points in $\mathbb{R}^m$ from the two classes are linearly separable by a hyperplane perpendicular to a suitable α. SVM seeks an α which maximizes the distance of any training point from the hyperplane; this can be written as:

$$\begin{array}{lll} \text{Minimize} & \frac{1}{2}\,\alpha\cdot\alpha \qquad (= \frac{1}{2}\|\alpha\|^2) & (4) \\ \text{subject to} & c_i(\alpha\cdot d_i + b) \geq 1 \quad \forall i = 1,\ldots,n, & \end{array}$$

where $\{d_1,\ldots,d_n\}$ are the training document vectors and $\{c_1,\ldots,c_n\}$ their corresponding classes. (We want an α such that $\text{sign}(\alpha\cdot d_i + b) = c_i$, so that their product is always positive.) The distance of any training point from the optimized hyperplane (called the *margin*) will be at least $1/\|\alpha\|$.

To handle the general case where a single hyperplane may not be able to correctly separate *all* training points, *fudge* variables $\{\xi_1,\ldots,\xi_n\}$ are introduced, and (4) enhanced as:

$$\begin{array}{llll} \text{Minimize} & \frac{1}{2}\alpha\cdot\alpha + C\sum_i \xi_i & & (5) \\ \text{subject to} & c_i(\alpha\cdot d_i + b) \geq 1 - \xi_i & \forall i = 1,\ldots,n, & \\ \text{and} & \xi_i \geq 0 & \forall i = 1,\ldots,n. & \end{array}$$

If d_i is misclassified, then $\xi_i \geq 1$, so $\sum_i \xi_i$ upper bounds the number of training errors, which is traded off against the margin using the tuned constant C. SVM packages solve the *dual* of (5), involving scalars $\lambda_1,\ldots,\lambda_n$, given by:

$$\begin{array}{lll} \text{Maximize} & \sum_i \lambda_i - \frac{1}{2}\sum_{i,j}\lambda_i\lambda_j c_i c_j (d_i\cdot d_j) & (6) \\ \text{subject to} & \sum_i c_i\lambda_i = 0 & \\ \text{and} & 0 \leq \lambda_i \leq C \quad \forall i = 1,\ldots,n. & \end{array}$$

The '$\frac{1}{2}$' in the objective (4) is not needed, but makes the dual look simpler. Having optimized the λs, α is recovered as

$$\alpha_{\text{SVM}} = \sum_i \lambda_i c_i d_i. \qquad (7)$$

If $0 < \lambda_i \leq C$, d_i is a "support vector". b can be estimated as $c_j - \alpha_{\text{SVM}}\cdot d_j$, where d_j is some document for which $0 < \lambda_j < C$. One can *tune C* and b based on a held-out validation data set and pick the values that gives the best accuracy. We will refer to such a tuned SVM as **SVM-best**.

Formula (6) represents a quadratic optimization problem. SVM packages iteratively refine a few λs at a time (called the *working set*), holding the others fixed. For all but very small training sets, we cannot precompute and store all the inner products $d_i\cdot d_j$. As a scaled-down example, if an average document costs 400 bytes in RAM, and there are only $n = 1000$ documents, the corpus size is 400000 bytes, and the inner products, stored as 4-byte `floats`, occupy $4\times 1000\times 1000$ bytes, ten times the corpus size. Therefore the inner products are computed on-demand, with a LRU cache of recent values, to reduce recomputation. In all SVM implementations that we know of, all the document vectors are kept in memory so that the inner products can be quickly recomputed when necessary.

2.4 Observations leading to our approach

It is natural to question the observed difference between the accuracy of NB classifiers and linear SVMs, given that they use the same hypothesis space (half-spaces). In assuming attribute independence, NB starts with a large *inductive bias* (loosely speaking, a constraint not guided by the training data) on the space of separating hyperplanes that it will draw from. SVMs do not propose any generative probability distribution for the data points and do not suffer from this form of bias. Another weakness of the NB classifier is that its parameters are based only on sample means; it takes no cognizance of variance. Fisher's discriminant does take variance into account (see Figure 2 later).

A linear SVM carefully finds a single discriminative hyperplane. Consequently, instances projected on the direction $\vec{\alpha}_{\text{SVM}}$ normal to this hyperplane show large to perfect inter-class separation. Intuitively, our hope is that we can be slightly sloppy (compared to SVMs) with finding discriminative direction(s), provided we can quickly find a number of such directions which can *collectively* help a decision tree learner separate the classes in the projected space. To achieve speed and scalability, it must be possible to cast our computation in terms of efficient sequential scans over the data with a small number of accumulators collecting sufficient statistics [7].

3 The proposed algorithm

Our proposed training algorithm has the broad outline shown in Figure 1. The important steps are **hill-climbing** and **remov**ing instances from the training set, which we will explain and rationalize shortly. Testing is simple, and essentially as fast as NB or SVM. We preserve the linear discriminants $\alpha^{(0)},\ldots,\alpha^{(k-1)}$ as well as the decision tree (in practice we found $k = 2\ldots 3$ to be adequate). Given a test document d, we find its k-dimensional representation $(d\cdot\alpha^{(0)},\ldots,d\cdot\alpha^{(k-1)})$ and submit this vector to the decision tree classifier, which outputs a class.

3.1 The hill-climbing step

Let the points with $c = -1$ ($c = 1$) be X (Y). Fisher's linear discriminant is a (unit) vector α such that the positive and negative training instances, projected on the direction α, are as "well-separated" as possible. The separation $J(\alpha)$,

```
$i \leftarrow 0$
initialize $D$ to the set of all training documents
**while** $D$ has at least one positive and one negative instance **do**
    initialize $\alpha^{(i)}$ to the vector direction joining the positive class centroid to the negative class centroid
    do **hill-climbing** to find a good linear discriminant $\alpha^{(i)}$ for $D$
    orthogonalize $\alpha^{(i)}$ w.r.t. $\alpha^{(0)}, \ldots, \alpha^{(i-1)}$ and scale it so that its $L_2$ norm $\|\alpha^{(i)}\| = 1$
    **remove** from $D$ those instances that are correctly classified by $\alpha^{(i)}$
    $i \leftarrow i + 1$
**end while**
let $\alpha^{(0)}, \ldots, \alpha^{(k-1)}$ be the $k$ linear discriminant vectors found
**for** each document vector $d$ in the original training set **do**
    represent $d$ as a vector of its projections $(d \cdot \alpha^{(0)}, \ldots, d \cdot \alpha^{(k-1)})$
    train a decision tree classifier with the $k$-dimensional training vector
**end for**
```

Figure 1: The proposed multiple linear discriminant algorithm.

shown in equation (8), is quantified as the ratio of the square of the difference between the projected means to the sum of the projected variances.

In matrix notation, if μ_X and μ_Y are the means (centroids) and $\Sigma_X = (1/|X|) \sum_X (x - \mu_X)(x - \mu_X)^T$ and $\Sigma_Y = (1/|Y|) \sum_Y (y - \mu_Y)(y - \mu_Y)^T$ are the covariance matrices for point sets X and Y, the best linear discriminant can be found in closed-form, which has been used in pattern recognition:

$$\arg\max_{\alpha} J(\alpha) = \left(\frac{\Sigma_X + \Sigma_Y}{2} \right)^{-1} (\mu_X - \mu_Y), \quad (11)$$

provided the matrix inverse exists.

However, inverting the covariance matrix is impractical in the document classification domain, because it is too large, and very likely ill-conditioned. Moreover, inversion will discard sparsity. Instead, we use a gradient ascent or hill-climbing approach: we start from a reasonable starting value of α and repeatedly find $\nabla J(\alpha) = (\partial J/\partial \alpha_1, \ldots, \partial J/\partial \alpha_m)$. Denoting the numerator (respectively, denominator) in the rhs of (8) as $N(\alpha)$ (respectively, $D(\alpha) = D_X(\alpha) + D_Y(\alpha)$), we can easily write down $\partial N/\partial \alpha_k$, $\partial D_X/\partial \alpha_k$, and $\partial D_Y/\partial \alpha_k$, as shown in Figure 2. From these values we can easily derive the value of $\partial J/\partial \alpha_i$ for all i in the term vocabulary. Once we find $\nabla J(\alpha)$, we use the standard WH update rule with a learning rate η:

$$\alpha_{\text{next}} \quad \leftarrow \quad \alpha_{\text{current}} + \eta \, \nabla J(\alpha). \quad (12)$$

The gist is that we need to maintain the following set of accumulators as we scan the documents sequentially:

- $\sum_X x \cdot \alpha$ (scalar)
- $\sum_Y y \cdot \alpha$ (scalar)
- $\sum_X x_i$ for each i (m numbers)
- $\sum_Y y_i$ for each i (m numbers)
- $\sum_X x_i(x \cdot \alpha)$ for each i (m numbers)
- $\sum_Y y_i(y \cdot \alpha)$ for each i (m numbers)

together with the current α, which is another m numbers. The total memory footprint is only $5m + O(1)$ numbers ($20m$ bytes), where m is the size of the vocabulary. For our Dmoz data set (see §4), $m \approx 1.2 \times 10^6$, which means we need only about 24 MB of RAM. All the vectors have dense array representations, so the time for one hill-climbing step is exactly **linear** in the size of the input data. It is also easy to see that all the expressions in Figure 2 can be expressed as simple `GROUP BY` and aggregate operations.

3.2 Pruning the training set

After a suitable number of hill-climbing steps, we need to discard points in D which are "well-separated" by the current α. This is achieved by projecting all points in D along α, so that they are now points on the line (each marked with $c = 1$ and $c = -1$), and sweeping the line for a minimum-error position where

- Most points on one side have $c = 1$ and most points on the other side have $c = -1$, and
- The number of points on the 'wrong' side is the minimum possible.

It is easy to do this in one sort of an n-element array and one sweep with $O(1)$ extra memory, so the total time for identifying well-separated documents is $O(n \log n)$ (and the total space needed is $O(m + n)$).

Typically, each new α helps us discard over 80% of D. To avoid scanning through the original D for every hill-climbing pass, we write out the surviving

$$J(\alpha) = \frac{\overbrace{\left(\frac{1}{|X|}\sum_X x\cdot\alpha - \frac{1}{|Y|}\sum_Y y\cdot\alpha\right)^2}^{N}}{\underbrace{\frac{1}{|X|}\sum_X (x\cdot\alpha)^2 - \left(\frac{1}{|X|}\sum_X x\cdot\alpha\right)^2}_{D_X} + \underbrace{\frac{1}{|Y|}\sum_Y (y\cdot\alpha)^2 - \left(\frac{1}{|Y|}\sum_Y y\cdot\alpha\right)^2}_{D_Y}} \tag{8}$$

$$\frac{\partial N}{\partial \alpha_i} = 2\left(\frac{1}{|X|}\sum_X x\cdot\alpha - \frac{1}{|Y|}\sum_Y y\cdot\alpha\right)\left(\frac{1}{|X|}\sum_X x_i - \frac{1}{|Y|}\sum_Y y_i\right) \tag{9}$$

$$\frac{\partial D_X}{\partial \alpha_i} = \frac{2}{|X|}\left(\sum_X x_i(x\cdot\alpha) - \frac{1}{|X|}\left(\textstyle\sum_X x_i\right)\left(\textstyle\sum_X x\cdot\alpha\right)\right) \tag{10}$$

Figure 2: The main equations involved in the hill-climbing step.

documents in a new file, which then becomes our D for finding the next α.

The intuition behind this algorithm is quite simple: having found a discriminant α we should retain only those points which fail to be separated in that direction. Furthermore, orthogonalizing the set of αs reduces the correlation between the components of the k-dimensional representation of documents, thus helping the decision tree find simpler orthogonal cuts in the feature space.

3.3 Inducing a decision tree

We used two roughly equivalent decision tree packages using Quinlan's C4.5 algorithm: C4.5 itself (`http://www.cse.unsw.edu.au/~quinlan/`) and the decision tree package in WEKA [22] (which we simply call WEKA; also see `http://www.cs.waikato.ac.nz/~ml/weka/`). In our context, a decision tree seeks to partition a set of labeled points $\{d\}$ in a geometric space, where each d is a vector $(d_0, \ldots, d_{k-1})$.

The decision tree induction algorithm uses a series of guillotine cuts on the space, each of which is expressed as a comparison of some component d_i against a constant, such that each final rectangular region has only positive or only negative points. The hierarchy of comparisons induces the decision tree, whose leaves correspond to final rectangular regions. To achieve a recall-precision trade-off, just as we can tune the offset b for a SVM, we can assign different weights to positive ($c = 1$) and negative ($c = -1$) instances in WEKA.

A decision tree with 'pure' (single-class) leaves usually *overfits* the training data and does not generalize well to held-out test data. Better generalization is achieved by *pruning* the tree, trading off the complexity of the tree with the impurity of the leaf rectangles in terms of mixing points belonging to different classes. This does not work too well for large m, which is why decision trees induced on raw text show poor accuracy. This is also why our dimensionality reduction via projection pays off well.

4 Experiments

The core of SIMPL (excluding document scanning and preprocessing) was implemented in only 600 lines of C++ code, making generous use of ANSI templates. The core of C4.5 is roughly another 1000 lines of C code. (In contrast, SVMLight, a very popular SVM package, is over 6000 lines.) "`g++ -O3`" was used for compilation. Programs were run on Pentium3 machines with 500–700 MHz CPUs and 512–2048 MB of RAM.

Two versions of SVM are publicly available: Sequential Minimum Optimization (SMO) by John Platt [17, 4] and SVMLight by Thorsten Joachims [9]. We found SVMLight to be comparable or better in accuracy compared to published SMO numbers. Our experiments are based on SVMLight, evaluating it for C between 1 and 60 (SVMLight's default is 21.37). We also evaluate several values of b in a suitable band around the separator. Other settings and flags are left at default values except where noted.

We use a few standard accuracy measures. For the following contingency table:

(Number of documents)		Estimated class	
		$\bar{c}$	c
Actual class	$\bar{c}$	n_{00}	n_{01}
	c	n_{10}	n_{11}

recall and precision are defined as $R = n_{11}/(n_{11} + n_{10})$ and $P = n_{11}/(n_{11} + n_{01})$. $F_1 = 2RP/(R + P)$ is also a commonly-used measure. A classifier may have parameters using which one can trade off R for P or vice versa. When these parameters are adjusted to get $R = P$, this value is called the "break-even point".

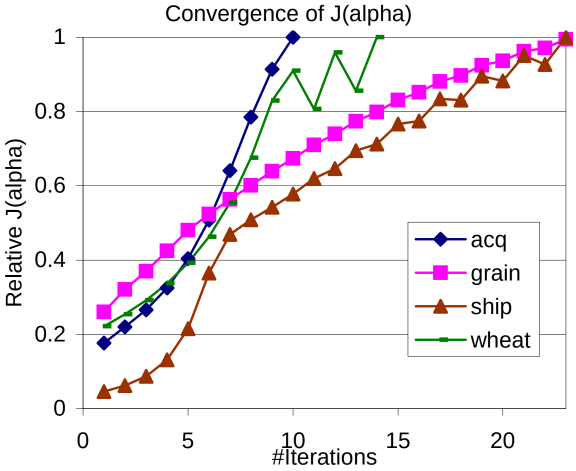

Figure 3: Hill-climbing to maximize $J(\alpha)$ is fast.

We use the standard "TFIDF" document representation from IR. In keeping with some of the best systems at TREC (`http://trec.nist.gov/`), our IDF for term t is $\log(|D|/|D_t|)$ where D is the document collection and $D_t \subseteq D$ is the set of documents containing t. The term frequency $\mathrm{TF}(d,t) = 1 + \ln\bigl(1 + \ln(n(d,t))\bigr)$, where $n(d,t) > 0$ is the raw frequency of t in document d (TF is zero if $n(d,t) = 0$). d is represented as a sparse vector with the tth component being $\mathrm{IDF}(t)\,\mathrm{TF}(d,t)$. The L_2 norm of each document vector is scaled to 1 before submitting to the classifier.

We use the following data sets. The first three are well-known in recent IR literature, small in size and suitable for controlled experiments on accuracy and CPU scaling. The last two data sets are large; they were mainly used to test memory scaling (but we verified that they show similar patterns of accuracy as the smaller data sets).

Reuters: About 7700 training and 3000 test documents ("MOD-APTE" split), 30000 terms, 135 categories. The raw text takes about 21 MB.

20NG: About 18800 total documents organized in a directory structure with 20 topics. For each topic the files are listed alphabetically and the first 75% chosen as training documents. There are 94000 terms. The raw concatenated text takes up 25 MB.

WebKB: About 8300 documents in 7 categories. About 4300 pages on 7 categories (faculty, project, etc.) were collected from 4 universities and about 4000 miscellaneous pages were collected from other universities. For each classification task, any one of the four university pages are selected as test documents and rest as training documents. The raw text is about 26 MB.

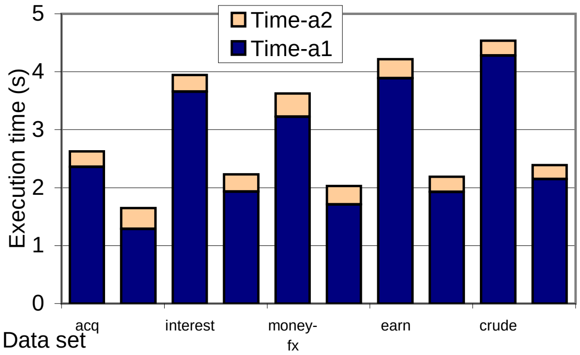

Figure 4: For each topic, the first bar shows time to compute $\alpha^{(0)}$ to convergence, then the time to compute $\alpha^{(1)}$ to convergence. The second bar shows the time to compute $\alpha^{(0)}$ sloppily, followed by computing $\alpha^{(1)}$ to convergence. Because earlier αs take more time to compute, we may even save overall time.

OHSUMED: 348566 abstracts from medical journals, having around 230000 terms and 308511 topics. The raw text is of size 400 MB. The first 75% are selected as training documents and the rest are test documents.

Dmoz: A cut was taken across the Dmoz (`http://dmoz.org/`) topic tree yielding 482 topics covering most areas of Web content. About 300 training documents were available per topic. The raw text occupied 271 MB.

The last two data sets start approaching the scale we envisage for real applications.

All our data sets sport more than two labels. For each label (e.g., 'cocoa'), a two-class problem ('cocoa' vs. 'not cocoa') is formulated. All tokens are turned to lowercase and standard SMART stopwords (`ftp://ftp.cs.cornell.edu/pub/smart/`) are removed, but no stemming is performed. No feature selection is used prior to running any of our classification algorithms: the naive Bayes classifier in the Rainbow library [13] (with Laplace and Lidstone's methods evaluated for parameter smoothing), SVMLIGHT and SIMPL. Alternatively, one may preprocess the collection through a common feature selector and then submit them to each classifier, which adds a fixed time to each classifier.

4.1 Accuracy

The hill-climbing approach is fast and practical. We usually settle at a maximum within 15–25 iterations: Figure 3 shows that $J(\alpha)$ quickly grows and stabilizes with successive iterations. Our default 'convergence' policy is to repeat hill-climbing until the increase in $J(\alpha)$ is less than 5% over 3 successive iterations. Such a policy guards against mild problems of local maxima and overshoots in case the learning rate η in equation (12), which is set to 0.1 throughout, is slightly off. This condition usually manifests itself in small oscillations in $J(\alpha)$, e.g., for topics *ship* and *wheat*. Our results are insensitive, within a wide range, to the specific choices of all these parameters, as the following experiment will also show.

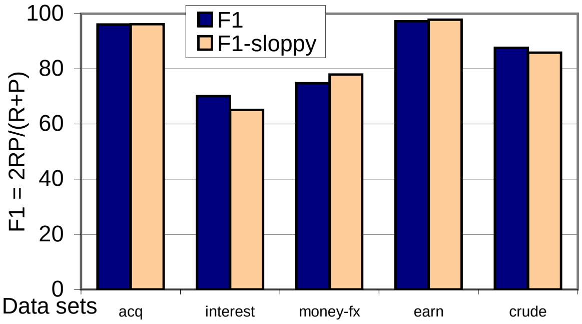

Figure 5: Quitting hill-climbing early has a modest impact on F_1 accuracy; in fact, in some cases, accuracy *improves*.

An interesting feature of SIMPL is that later αs can compensate for some sloppiness in finding an earlier α. For several problems, we first optimized $\alpha^{(0)}$ to convergence and noted the number of iteration required. (We also found $\alpha^{(1)}$ etc.) Then we reran the experiment with only *half* as many iterations for $\alpha^{(0)}$, which was therefore sub-optimal. We had to pay for this sloppiness during the estimation of subsequent αs (Figure 4), but because the training data has shrunk significantly, this actually *saves* us time.

In Figure 5 we note that even though terminating the hill-climbing for $\alpha^{(0)}$ halfway through saves almost *half* the training time, it entails little loss in accuracy. In fact, in some cases, because one of precision and recall falls less than the other, we may even *gain* accuracy in F_1 terms. Such resilience to variation in policy and parameters is very desirable.

We also show some scatter-plots of training and test data projected along $(\alpha^{(0)}, \alpha^{(1)})$ by SIMPL, shown in Figure 6. This is for the relatively difficult topic *money-fx* in the Reuters data, which a linear SVM could not separate. We see that the $\alpha^{(0)}$ direction is already quite effective for class discrimination, and most points in the confusion zone of $\alpha^{(0)}$ are effectively separated by $\alpha^{(1)}$ in the 2d map. $\alpha^{(1)}$ successfully clusters most of the negative class in the upper portion, while the positive class is more evenly spread out on the y-axis. These observations support Joachim's experience that the VC-dimension of many text classification benchmarks is very low.

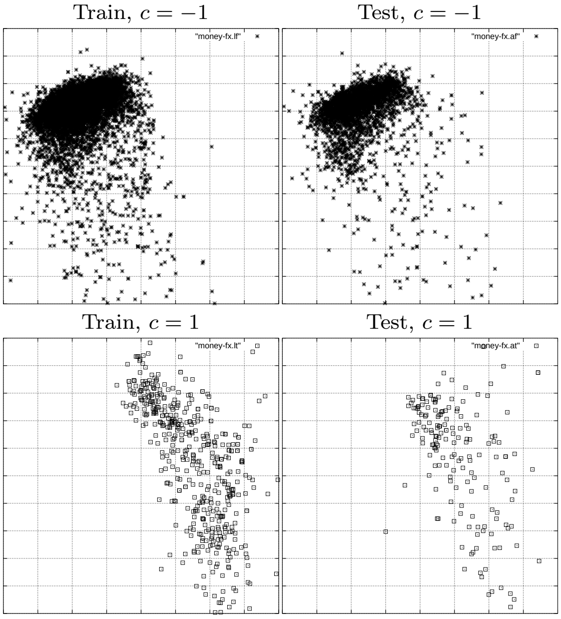
Train, $c = -1$ Test, $c = -1$
Train, $c = 1$ Test, $c = 1$

Figure 6: Training and test data projected on to the first two αs found by SIMPL, hinting that high-accuracy decision trees can be induced from the reduced-dimension data.

Another reason for SIMPL's high speed is that typically, we can lop off over 80% of the current training set for every additional $\alpha^{(i)}$ that we find. For example, for the Dmoz data, the numbers of surviving documents were 116528, 7074, 230, and 2 before finding $\alpha^{(0)}, \alpha^{(1)}, \alpha^{(2)}$ and $\alpha^{(3)}$ respectively. Consequently, SIMPL generates only 2–4 linear projections before running out of training documents. How many projections are needed to retain enough information for C4.5 to achieve high accuracy? Figure 7 shows the effect of including up to the first three αs. In all the cases, the first *two* αs are sufficient for peak accuracy. The general trend is that precision and recall approach each other as we include more projections, improving F_1. The loss in one is more than made up by the gain in the other. This result also shows that we can improve beyond linear regression (§2.2) by using additional projections. It is also reassuring that including more αs (up to 5) than necessary never hurt accuracy.

How does SIMPL measure up against SVM overall? Figure 8 shows a bird's-eye view of all data sets and all algorithms: it compares the F_1 score for naive Bayes (NB), SIMPL, SVM, and SVM-best (see §2.3). SIMPL beats NB in 33 out of 35 cases. (Lidstone smoothing had only mild effects on NB accuracy.) SIMPL beats SVM in 23 out of 35 cases. On an average we are a few percent

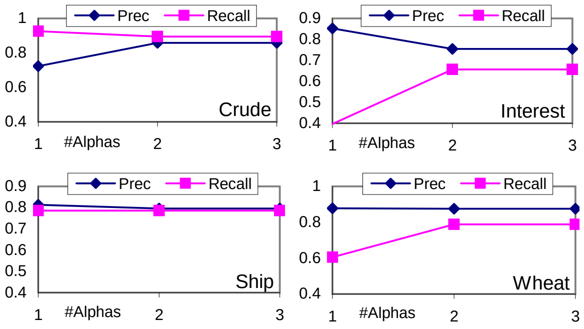

Figure 7: Variation of precision and recall with the number of linear projections used by C4.5. Two projections are almost always adequate.

short of SVM-best, but there are several cases (marked by stars) where we beat even SVM-best. (The extent to which tuning the offset parameter b improved SVM-best beyond SVM surprised us, and is not reported elsewhere. Tuning C had less effect.) We note that this is a comprehensive study over many diverse, standard data sets, and the high accuracy of SIMPL is very stable across the board.

F_1, being the harmonic mean, favors algorithms whose recall is close to precision, which is the case with SIMPL. To be fair, a closer look shows that SIMPL usually loses to SVM-best by a small margin in either recall or precision, but beats it in the other (Figure 9. In a few cases we beat SVM in *both* recall and precision. This is possible because we are not limited to a single planar separator. Because we use a decision tree in the projected space, we can learn, say a function like EXOR which a linear SVM cannot. Although SVMs with more complex kernels may be used, they are slower to train than linear SVMs.

We also compared our accuracy with that of C4.5 run directly on the raw text, reported in earlier work [4, 8]. We see that although we too use C4.5, our accuracy is substantially better, thanks to our novel feature combination and transformation steps. In addition, SIMPL runs much faster than C4.5 on the raw data.

Finally, we show a scatter-plot of F_1 scores against the positive class *skew* (ratio of the number of documents to the number of documents with $c = 1$) in Figure 10. While all methods suffer somewhat from skew, clearly NB suffers most and SIMPL suffers least.

4.2 Performance

Having established that our accuracy is comparable to SVM, we turn to a detailed investigation of the scalability and IO behavior of the two

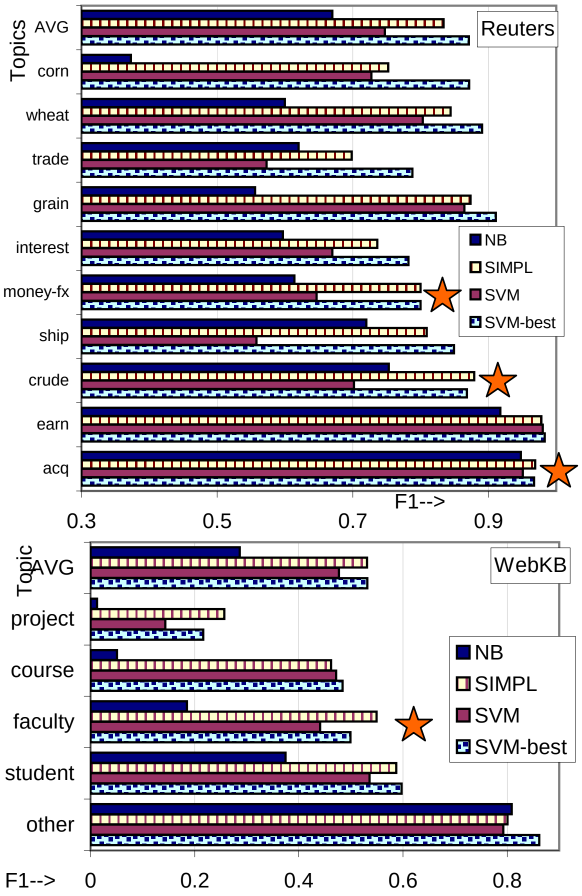

Figure 8: Aggregate F_1 scores for NB, SIMPL, and SVM for some of the most populated topics of the Reuters and WebKB data sets. We decisively beat NB (by a 15–20% margin) in most cases. We also frequently beat SVM (average 6% margin), and lose to SVM-best by a narrow margin (average 3%).

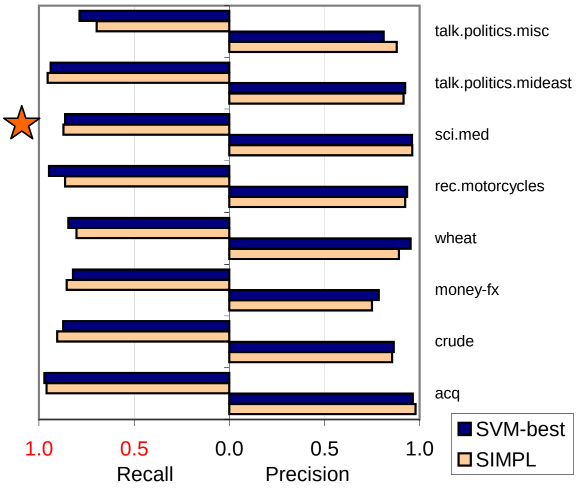

Figure 9: Most often we lose to SVM-best by a small margin in one of recall and precision and beat it in the other. Stars mark where we win in both.

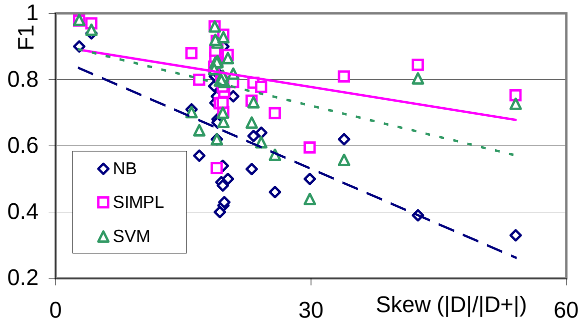

Figure 10: SIMPL shows the least adverse effect of class skew on F_1 accuracy among naive Bayes (NB), SIMPL, and SVM.

algorithms. We will first compare the scaling of CPU time with training set size, assuming enough RAM is available.

We did not include the initial time required to turn the raw training documents into the compact representation required for sequential scans in the case of SIMPL and the LRU cache required by SVM. We started timing the algorithms only after these initial disk structures were ready. The time consumed for preprocessing depends on a host of non-standard factors, such as the raw representation and system policies: single vs. many files, stopword detection and word truncation, term weighting, etc. We used the OHSUMED and Dmoz data for performance measurements. We report on Dmoz, OHSUMED being broadly similar.

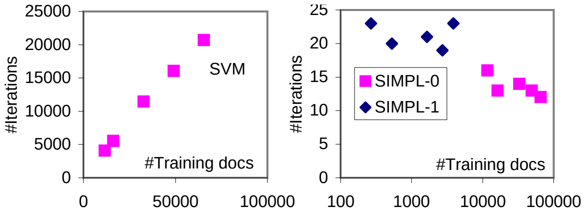

Figure 11: Number of iterations needed by one run of SVM and two runs of SIMPL (one finding $\alpha^{(0)}$, the other, $\alpha^{(1)}$). The number of SVM iterations seem to scale up with the problem size, unlike SIMPL.

We first observe in Figure 11 that, unlike SVM, the number of iterations needed for our hill-climbing step is largely independent of the number of training documents. The time taken by single hill-climbing iteration is linear in the total input size, defined as $\sum_d |\{t \in d\}|$, plus $O(n \log n)$ for n documents. Because $\log n$ is small and the time for sorting is very small compared to the α update step, the total time for SIMPL is expected to be essentially linear.

This is confirmed in the log-log plot shown in Figure 12, where the least-square fit for SIMPL is roughly $t \propto n^{0.955}$, where t is the execution time. In contrast, the regression for the running time of SVM is $t \propto n^{1.88}$, which corroborates earlier measurements by Platt, Joachims, and others. This difference translates to a running time ratio (SVM to SIMPL) of almost two orders of magnitude for n as small as 65000. For collections at the scale of Yahoo! or the full Dmoz data set (millions of documents) the ratio will reach several orders of magnitude.

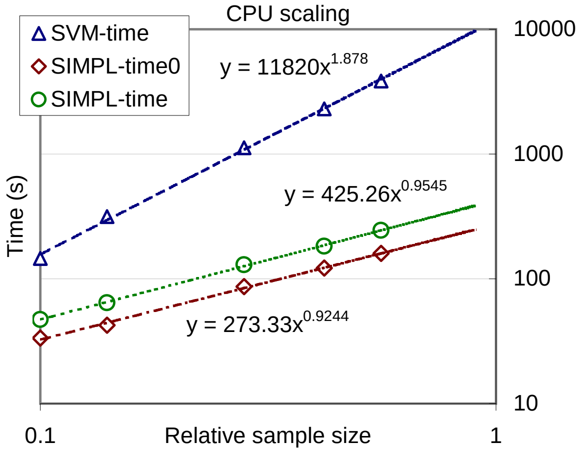

Figure 12: Scaling of overall CPU time, excluding preprocessing time, with training set size for SVM and SIMPL, keeping all training data in memory. The line marked "SIMPL-time0" shows the time for finding just the first α, and the line marked "SIMPL" shows the total time for SIMPL. A sample of 65000 documents were chosen from Dmoz.

All public SVM implementations that we know of, including SVMLIGHT and SMO, load all the training document vectors into memory. With limited memory, we expect the performance gap between SVM and SIMPL to be even larger. In our final set of experiments, we study the behavior of SVM with limited memory.

As mentioned before (§2.3), SVM optimizes the λs corresponding to a small working set of documents at a time. In SVMLIGHT, the size of this working set (typically 10–50) is set by the '`-q`' option. There are standard heuristics for picking the next working set to speed up convergence, but applying these may replace the entire working set, reducing cache locality. SVMLIGHT provides another option '`-n`' which limits the number of new λs that are permitted to enter the working set in each iteration. Reducing this increases cache locality, but can lead to more iterations. A sample of this trade-off is shown in Figure 13. Even though a heavy replacement rate decreases cache hits and increases the time per iteration, the number of iterations is cut so drastically that a large value for '`-n`' is almost always a better choice.

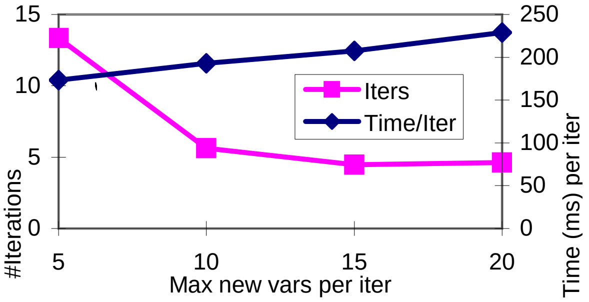

Figure 13: Although cache misses and time per iteration increase if the number of new variables optimized per iteration is capped, the drastic savings in the number of iterations reduces overall time. The corpus and cache sizes were 24000 and 2000 documents and '`-q 20`' was used.

How do cache hits and misses translate into running time overheads? This can be measured either by limiting physical memory available to the computer (in Linux, using a line of the form `append="mem=512M"` in `/etc/lilo.conf`), or by letting SVM do its own document cache management and instrumenting this cache. The former option is appealing to the end-user. However, system processes, OS buffer cache, device interrupts and paging make measurements unreliable. Moreover, the OS cache is physical and cannot exploit the structure of the SVM computation. Therefore we expect large-scale SVM packages to implement their own caching. We used a disk without any file system as a raw block device (an ATA/66 7200RPM Seagate-ST330630A drive with 2 MB on-board cache on `/dev/hdc1`), which precluded interference owing to OS buffering. We built a LRU cache on it with a preconfigured main memory quota. Servicing a miss usually involved exactly one seek on disk unless the data was in the disk's on-board cache.

Figure 14 shows a break-up of times spent in computation (CPU), hit, miss and eviction processing. This sample from Dmoz had 23428 documents (23 MB of sparse term vectors) with 360000 features, so SIMPL needs only 6 MB of RAM and runs for only 80 seconds with close-to-100% CPU utilization. In contrast, if SVM is given 6 MB of cache (about 6000 documents), it takes over 1300 seconds, of which 60% is spent in servicing evictions and misses.

Extending from the small-scale experiment above, we present in Figure 15 a large-scale experiment where we go from a 10% to a 100% sample of our Dmoz data (117920 documents, 110 MB in RAM). For each sample we determine the amount of RAM needed by SIMPL, and give that quantity of cache to SVM, and compare the running times. The graph is superficially similar to Figure 12, but a closer look shows that the ratio of SVM to SIMPL running times is larger owing to cache overheads. Summarizing, SIMPL beats SVM w.r.t. both CPU and cache performance, but the near-quadratic CPU scaling of SVM makes cache overheads appear less serious than they really are: the *total* time spent by SIMPL is less than 20% of the time spent by SVM on cache management alone.

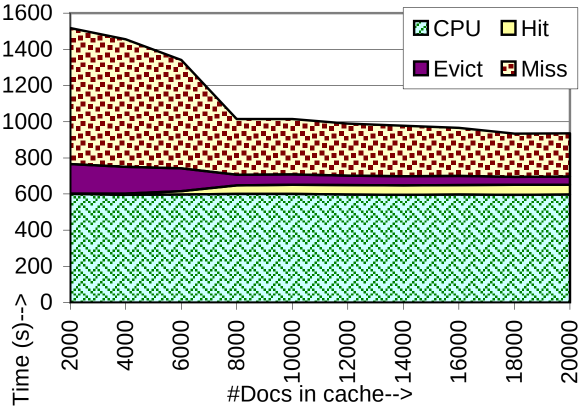

Figure 14: CPU time, cache hit time, miss time and eviction time plotted against the relative size of cache available to SVM.

5 Conclusion

We have presented SIMPL, a new classifier for high-dimensional data such as text. SIMPL is very simple to understand and easy to implement. SIMPL is very fast, scaling linearly with input size, as against SVM, which shows almost quadratic scaling. SIMPL uses efficient sequential scans over the training data, unlike SVM, which has an inferior disk and cache access pattern and locality of reference. This performance boost carries little or no penalty in terms of accuracy: we often beat SVM in the F_1 measure, and closely

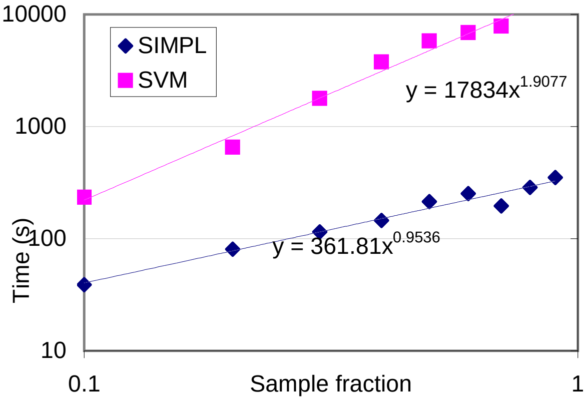

Figure 15: When SVM and SIMPL are given the same amount of RAM as we scale up the size of the training set, SIMPLperforms far better.

match SVM in recall and precision. SIMPL beats naive Bayes and decision tree classifiers decisively for text learning tasks. We also present a detailed study of the IO behavior of SVM, which shows that, in contrast to SIMPL's efficient sequential scans, SVM's cache locality is relatively poor.

Our work shows that even though SVMs are elegant, powerful, and theoretically appealing, they have not rendered the search for practical and IO-efficient alternatives unnecessary or fruitless. A natural area of future work is to identify properties of data sets which guarantee near-SVM accuracy using SIMPL. Another area of applied work is to test SIMPL vis-a-vis non-linear SVM for non-textual training data with somewhat higher VC-dimension.

Acknowledgments: Thanks to Pedro Domingos for helpful discussions, Thorsten Joachims for generous help with SVMLIGHT, Kunal Punera for help with preparing some data sets, and Shantanu Godbole for helpful comments on the manuscript.

References

[1] R. Agrawal, R. J. Bayardo, and R. Srikant. Athena: Mining-based interactive management of text databases. In *7th International Conference on Extending Database Technology (EDBT)*, Konstanz, Germany, Mar. 2000. Online at `http://www.almaden.ibm.com/cs/people/ragrawal/papers/athena.ps`.

[2] T. Cooke. Two variations on Fisher's linear discriminant for pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 24(2):268–273, feb 2002. `http://www.computer.org/tpami/tp2002/i0268abs.htm`.

[3] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

[4] S. Dumais, J. Platt, D. Heckerman, and M. Sahami. Inductive learning algorithms and representations for text categorization. In *7th Conference on Information and Knowledge Management*, 1998. Online at `http://www.research.microsoft.com/~jplatt/cikm98.pdf`.

[5] P. Frankl and H. Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory*, B 44:355–362, 1988.

[6] G. Fung and O. L. Mangasarian. Proximal support vector classifiers. In F. Provost and R. Srikant, editors, *Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 77–86, San Francisco, CA, aug 2001. ACM. Also published as Data Mining Institute Technical Report 01-02, University of Wisconsin, February 2001, see `http://www.cs.wisc.edu/~gfung/`.

[7] G. Graefe, U. M. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Knowledge Discovery and Data Mining*, volume 4, pages 204–208, New York, 1998. AAAI Press.

[8] T. Joachims. Text categorization with support vector machines: learning with many relevant features. In C. Nédellec and C. Rouveirol, editors, *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398 in LNCS, pages 137–142, Chemnitz, DE, 1998. Springer Verlag, Heidelberg, DE.

[9] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*. MIT Press, 1999. See `http://www-ai.cs.uni-dortmund.de/DOKUMENTE/joachims_99a.pdf`.

[10] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM Symposium on Theory of Computing*, pages 599–608, 1997.

[11] D. D. Lewis, R. E. Schapire, J. P. Callan, and R. Papka. Training algorithms for linear text classifiers. In H.-P. Frei, D. Harman, P. Schäuble, and R. Wilkinson, editors, *Proceedings of SIGIR-96, 19th ACM International Conference on Research and Development in Information Retrieval*, pages 298–306, Zürich, CH, 1996. ACM Press, New York, US.

[12] O. L. Magasarian and D. R. Musicant. Lagrangian support vector machines. Technical Report 00-06, Data Mining Institute, University of Wisconsin, Madison, June 2000. Online at `http://www.cs.wisc.edu/~musicant/`.

[13] A. McCallum. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. Software available from `http://www.cs.cmu.edu/~mccallum/bow/`, 1998.

[14] A. McCallum and K. Nigam. A comparison of event models for naive Bayes text classification. In *AAAI/ICML-98 Workshop on Learning for Text Categorization*, pages 41–48. AAAI Press, 1998. Also technical report WS-98-05, CMU; online at `http://www.cs.cmu.edu/~knigam/papers/multinomial-aaaiws98.pdf`.

[15] S. K. Murthy, S. Kasif, and S. Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

[16] D. Pavlov, J. Mao, and B. Dom. Scaling-up support vector machines using boosting algorithm. In *International Conference on Pattern Recognition (ICPR)*, Barcelona, Sept. 2000. See `http://www.cvc.uab.es/ICPR2000/`.

[17] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, 1998. Online at `http://www.research.microsoft.com/users/jplatt/smoTR.pdf`.

[18] R. E. Schapire. The boosting approach to machine learning: An overview. In *MSRI Workshop on Nonlinear Estimation and Classification*, Berkeley, CA, Mar. 2001. See `http://stat.bell-labs.com/who/cocteau/nec/` and `http://www.research.att.com/~schapire/boost.html`.

[19] H. Schutze, D. A. Hull, and J. O. Pederson. A comparison of classifiers and document representations for the routing problem. In *SIGIR*, pages 229–237, 1995. Online at `ftp://parcftp.xerox.com/pub/qca/SIGIR95.ps`.

[20] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *VLDB*, pages 544–555, 1996.

[21] A. Shashua. On the equivalence between the support vector machine for classification and sparsified Fisher's linear discriminant. *Neural Processing Letters*, 9(2):129–139, 1999. Online at `http://www.cs.huji.ac.il/~shashua/papers/fisher-NPL.pdf`.

[22] I. H. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, San Francisco, oct 1999. ISBN 1-55860-552-5.