Fast and accurate text classification via multiple linear discriminant projections

Soumen Chakrabarti, Shourya Roy, Mahesh V. Soundalgekar

IIT Bombay

Edited by Y. Ioannidis

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 that are valuable for mapping billions of Web pages into Yahoo!-like directories. We present SIMPL, a nearly linear-time classification algorithm that 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 the running time of a popular SVM implementation by orders of magnitude. While describing SIMPL, we make a detailed experimental comparison of SVM-generated discriminants with Fisher's discriminants, and we also report on an analysis of the cache performance of a popular SVM implementation. Our analysis shows that SIMPL has the potential to be the method of choice for practitioners who want the accuracy of SVMs and the simplicity and speed of naive Bayes classifiers.

Keywords: Text classification – Discriminative learning – Linear discriminants

1 Introduction

Text classification is a well-studied problem in document management. A *classifier* or *learner* is first presented with *training* documents d, each assigned a label, c, drawn from two possible labels: $+1$ or -1. Depending on the application, the label may indicate some property of the document, e.g., whether a news article is about sustainable energy or whether an email is a "spam."

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 choose one of the two labels for each test document.

If an application demands more than two labels (e.g., a Web directory with 15 broad topics at the top level), it is common to train one learner for each topic γ; documents marked with γ are labeled $+1$ and all other documents are labeled -1 [8, 14]. This is called "one-vs.-rest" classification.

Text classification has numerous potential applications including the automatic maintenance of topic directories such as the Open Directory (also called Dmoz, see `http://dmoz.org`) and Yahoo! (`http://www.yahoo.com`), filtering email for spam [31], and collaborative filtering [2]. Naive Bayes (NB) [3], maximum entropy (maxent) [28] and support vector machines (SVMs) [8,14,37] are some of the best-known classifiers employed to date on text data.

Not surprisingly, there is a trade-off between simplicity and accuracy. NB classifiers are simple to understand and easy to implement, access disk-resident data efficiently, and run fast, but they may show mediocre accuracy. SVMs are among the most accurate classifiers known for text applications: they beat NB accuracy by a decisive margin and are generally better than maxent classifiers. NB classifiers also tend to score lower than maxent classifiers in terms of accuracy. Such accuracy differences are intriguing because SVM, maxent, and NB classifiers all learn a hyperplane that separates the positive examples ($c = 1$) from the negative ones ($c = -1$), documents being represented as vectors in a high-dimensional term space. SVM and maxent undertake complex nonlinear numeric optimizations (which are highly nontrivial to understand and implement) to search for a high-quality separator, whereas NB makes a quick but generally inferior choice.

NB takes time essentially linear in the number n of training documents [25,26], whereas SVMs take time proportional to n^a, where a is typically between 1.8 and 2.1. Thanks to some clever implementations [15,30], SVMs have been trained on several thousand instances despite their near-quadratic complexity. However, scaling up to hundreds of thousands of instances appears infeasible at this time. Memory footprint is

another issue; several popular SVM packages store training vectors in memory. (Some exceptions are noted in Sect. 1.2.)

Scalability and memory footprint can become critical issues as enormous training sets become increasingly available. Web directories such as the Open Directory and Yahoo! contain millions of training instances that occupy tens of gigabytes, whereas even high-end servers are mostly limited to 1–2 GB of RAM. Given the sparsity of data in the text domain, sampling the training data is dangerous because the sample may exclude thousands of useful features. Joachims [14] shows that a large fraction of terms reveal at least some useful class information, and therefore every additional training document could potentially be a source of features. We confirm this in Sect. 4.6, where we see that the ability to scale better also translates to a better accuracy-time trade-off.

In summary, despite the theoretical elegance and superiority of SVMs, their IO behavior and CPU scaling are important concerns. There is a need for easy-to-implement text classifiers that match the simplicity and efficiency of NB classifiers while giving an accuracy comparable to SVMs.

1.1 Our contribution

We design, implement, and evaluate a new, simple text classification algorithm that requires 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 [17, Sect. 11.5] 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; it makes only fast sequential scans over the input; its CPU time is almost 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.[1] 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. Although there is no theoretical bound on the number of linear projections SIMPL may need, only two to three projections are usually enough to achieve high accuracy.

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 [34]. In the last few years, the machine learning and text mining communities have evolved other powerful classifiers, such as SVMs and maxent classifiers. 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 a popular SVM implementation accessing documents from a LRU cache having limited size. If the SVM implementation 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 the SVM implementation 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 [9] 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 interpoint distances with high probability. Later work has established random projection as a valuable general technique for dealing with high-dimensional data. Kleinberg projected these points to $\Theta(m \log^2 m)$ randomly directed lines to answer approximate nearest-neighbor queries efficiently [18]. Dasgupta [5,6] used random projection to learn a mixture of Gaussians, showing en route that well-separated Gaussians remain well separated upon projection.

For a classification task, we need not preserve *all* distances carefully. We simply need a subspace that separates the positive and negative instances well (a special case of "projection pursuits" [10,36]). In an early study by Schütze, Hull, and Pedersen [33], even *single* linear discriminants compared favorably with neural networks for the document routing problem. Lewis et al. [22] reported accurate prediction using a variety of regression strategies for good (single) linear predictors. The recent success of linear 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 Sect. 2.4) found by a SVM [35]. 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 [7] and SVMs, which we discuss in detail in Sect. 2 and Sect. 3. Independently, Cooke [4] has suggested discarding well-separated training points before finding Fisher's linear discriminant but has not used multiple projections to generate a surrogate representation to be used by a more powerful learning algorithm such as a decision tree.

Many researchers have worked on reducing the memory footprint and running time of SVM optimizations. One strategy, pursued by Mangasarian and coworkers, is to change the objective function slightly, which enables use of more efficient mathematical programming machinery without affecting the utility of the solution in practice. Lagrangian SVM [24], proximal SVM [11], and incremental SVM [12] are examples of this paradigm; an incremental SVM can in fact retire and add new training data efficiently. These SVM variants involve invert-

[1] SIMPL is available at `http://www.cse.iitb.ac.in/~soumen`.

ing an $m \times m$ matrix (m is the number of dimensions), which is readily done for moderately large values of m (hundreds to thousands) but demands too much main memory ($O(m^2)$) and too much time ($O(m^3)$) in the text domain, where $m > 70000$ is not uncommon. The inverted matrix is generally not sparse. However, two other techniques from this family, successive over-relaxation SVM [23] and reduced SVM (a sampling technique) [20], may compare favorably with SIMPL (apart from being theoretically more elegant). Pavlov, Mao, and Dom have developed a different sampling technique [29].

SIMPL may be interpreted as an approximation to *boosting* [32]. Boosting seeks to improve a "weak" learner (which makes decisions only slightly better than random guessing) by running it many times on successively altered training distributions. The first training distribution is generated from the training data by assigning equal probability $1/n$ to each instance. Subsequent distributions are generated by boosting the probability of instances that the weak learner labeled incorrectly. Each learner in the sequence also gets a score based on its error rate. The overall learner is a weighted majority of the set of weak learners, where the weights depend on the scores of the weak learners. SIMPL simply throws away correctly learned points but uses a more complex combination of the weak learners.

Our approach is also related to oblique decision trees (ODTs) [27], which try to find nonorthogonal hyperplane cuts in the decision-tree setting. 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 two). We also found SIMPL to be more accurate than orthogonal decision trees. A comparison with ODTs may be worthwhile if ODTs can be trained within reasonable time on high-dimensional data.

2 Preliminaries

A host of linear classifiers have been used for text classification. A document d is represented as a "feature vector" $\boldsymbol{d}$ with a component d_t for each term t in the vocabulary. We will overload d to mean $\boldsymbol{d}$ where there is no confusion. Generally, the more often t occurs in d, the larger the value of d_t. A linear classifier is characterized by a vector α and a scalar constant b, and it predicts the class of d as

$$c_{\text{guess}} = \text{sign}(\alpha \cdot d + b), \tag{1}$$

where $\cdot$ indicates a dot-product and $\text{sign}(x) = 0$ if $x = 0$, $\text{sign}(x) = 1$ if $x > 0$, and $\text{sign}(x) = -1$ if $x < 0$.

Linear classifiers work well for document classification along the lines of broad topics. A document about (the game of) cricket will tend to use terms like wicket, run, ball, and pitch frequently and will also tend to use them together, compounding the evidence that the document is about cricket. α and b together represent a hyperplane, which cuts across each axis (corresponding to a term) at some offset. The offset acts as a threshold: if the term occurs more frequently, the document is assigned the label $+1$. In general, if a linear combination of the frequencies of important terms exceeds a threshold, the document will lie on the "positive" ($c = 1$) side of the hyperplane. This discussion may help explain why linear discriminants are good at text classification.

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

$$\Pr(d|c, \ell_d) = \binom{\ell_d}{\{n(d,t)\}} \prod_{t \in d} \theta_{c,t}^{n(d,t)} \tag{2}$$

where $n(d,t)$ is the number of times t occurs in d and $\theta_{c,t}$ are suitably estimated [1,26] 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

$$\begin{array}{l} \log \Pr(c = 1) + \sum_{t \in d} n(d,t) \log \theta_{1,t} \quad \text{and} \\ \log \Pr(c = -1) + \sum_{t \in d} n(d,t) \log \theta_{-1,t} \end{array} \tag{3}$$

where $\Pr(c = \ldots)$, called the class *priors*, are the fractions of training instances in the respective classes. Simplifying Eq. 3, 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 Sect. 4) and "$\cdot$" denotes a dot-product.

2.2 Maximum entropy classifiers

Whereas Bayesian classifiers estimate class-conditional distributions $\Pr(d|c)$ for each class c, a maxent classifier directly estimates a parametric model for $\Pr(c|d)$. There is a model parameter $\mu_{c,t}$ for every class c and term t (as in NB). A commonly used parametric form of $\Pr(c|d)$ is

$$\Pr(c|d) \propto \prod_{t \in d} \mu_{c,t}^{n(d,t)/\sum_\tau n(d,\tau)} \tag{4}$$

Introducing a normalizing constant to make $\Pr(c|d)$ add up to 1 over all c, we get

$$\Pr(c|d) = \frac{1}{Z(d)} \prod_{t \in d} \mu_{c,t}^{n(d,t)/\sum_\tau n(d,\tau)} \tag{5}$$

Nigam et al. [28] discuss in detail how to optimize the parameters using training data. In the two-class case, using a maxent

classifier to classify a test document d amounts to comparing (after taking logs)

$$\sum_{t\in d} \frac{n(d,t)}{\sum_\tau n(d,\tau)} \log \mu_{1,t} \quad \text{and} \quad \sum_{t\in d} \frac{n(d,t)}{\sum_\tau n(d,\tau)} \log \mu_{-1,t}$$
$$\text{or} \quad \sum_{t\in d} n(d,t) \log \mu_{1,t} \quad \text{and} \quad \sum_{t\in d} n(d,t) \log \mu_{-1,t} \qquad (6)$$

again, clearly a linear discriminant.

2.3 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 information retrieval (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. Let each vector d be augmented by one extra element, always set to 1, and a corresponding extra dimension added to α, to simplify notation and get rid of b.The WH approach starts with some initial estimate $\alpha^{(0)}$ (with the extra dimension representing b), considers (d_i, c_i) one by one, and updates $\alpha^{(i-1)}$ to $\alpha^{(i)}$ as follows:

$$\alpha^{(i)} = \alpha^{(i-1)} - 2\eta(\alpha^{(i-1)} \cdot d_i - c_i)d_i \; . \qquad (7)$$

The final α used for classification is usually the average of all αs found along the way. Schütze, Lewis, et al. [22,33] have applied WH and other update methods (such as the exponentiated gradient method) to design high-accuracy linear classifiers for text, improving upon traditional Rocchio-style relevance feedback. 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.

2.4 Linear support vector machines

Like NB, linear SVMs (LSVMs) also make a decision by thresholding $\alpha_{\text{SVM}} \cdot d + b$ (the estimated class is $+1$ or -1 depending on whether 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 that maximizes the distance of any training point from the hyperplane; this can be written as:

$$\begin{aligned} &\text{Minimize} \;\; \tfrac{1}{2}\,\alpha \cdot \alpha \qquad (= \tfrac{1}{2}\|\alpha\|^2) \qquad (8)\\ &\text{subject to} \;\; c_i(\alpha \cdot d_i + b) \geq 1 \quad \forall i = 1, \ldots, n \end{aligned}$$

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 Eq. 8 is enhanced as:

$$\begin{aligned} &\text{Minimize} \quad \tfrac{1}{2}\alpha \cdot \alpha + C\textstyle\sum_i \xi_i \qquad (9)\\ &\text{subject to} \quad c_i(\alpha \cdot d_i + b) \geq 1 - \xi_i \quad \forall i = 1, \ldots, n\\ &\quad \text{and} \qquad \xi_i \geq 0 \qquad\qquad \forall i = 1, \ldots, n \end{aligned}$$

$\sum_i \xi_i$ is the "sum of violations" of the misclassified training points, which is traded off against the margin width $\frac{1}{2}\alpha \cdot \alpha$ using the tuned constant C.

SVM packages solve the *dual* of Eq. 9, involving scalars $\lambda_1, \ldots, \lambda_n$, given by:

$$\begin{aligned} &\text{Maximize} \; \textstyle\sum_i \lambda_i - \frac{1}{2}\sum_{i,j} \lambda_i\lambda_j c_i c_j (d_i \cdot d_j) \qquad (10)\\ &\text{subject to} \qquad \textstyle\sum_i c_i\lambda_i = 0\\ &\quad \text{and} \quad 0 \leq \lambda_i \leq C \quad \forall i = 1, \ldots, n \end{aligned}$$

Having optimized the λs, α is recovered as

$$\alpha_{\text{SVM}} = \textstyle\sum_i \lambda_i c_i d_i \qquad (11)$$

If $0 < \lambda_i \leq C$, then 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 $0 < \lambda_j < C$.

It is common to set C to $1/r^2$, where r is the radius of the smallest ball containing all training vectors, or the average Euclidean norm of the d-vectors. A fixed choice of C saves time but is rarely the best possible value in practice. In principle, the best value of b should fall out of the training process, but in practice, tuning b helps as well. Practitioners generally use a held-out validation data set to search over a large range of values for C and a smaller range of values for b, which means they have to run the (time-consuming) SVM induction program many, many times. It is natural to want to avoid this extra work, so we compare SIMPL with *two* versions of SVM: one in which $C = 1/r^2$ and b is not tuned, and another in which we search over C and b, which we call **SVM-best**.

Equation 10 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 an 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. (This observation excludes SVM variants that do not need quadratic programming, such as the ones discussed in Sect. 1.2.)

2.5 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 Fig. 2 below).

A linear SVM carefully finds a single discriminative hyperplane. Consequently, instances projected on the direction $\boldsymbol{\alpha}_{\mathrm{SVM}}$ normal to this hyperplane show large to perfect interclass separation. Intuitively, our hope is that we can be slightly sloppy (compared to SVMs) with finding discriminative directions, provided we can quickly find a number of such directions that 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 [13].

3 The proposed algorithm

Our proposed training algorithm has the broad outline shown in Figure 1. The outer while-loop of SIMPL finds several linear discriminants $\alpha^{(0)}, \alpha^{(1)}, \ldots$ one by one. To compute each α, we need a **hill-climbing** step. To prepare the stage for the next α, we need to **remove** some instances from the training set. We will explain and rationalize these steps in this section.

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 $J(\alpha)$, shown in equation (13), 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, and has been used in pattern recognition:

$$\arg\max_{\alpha} J(\alpha) = \left(\frac{\Sigma_X + \Sigma_Y}{2} \right)^{-1} (\mu_X - \mu_Y) \tag{12}$$

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 $\alpha = (\alpha_1, \ldots, \alpha_m)$ 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 Eq. 13 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 Fig. 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:

$$\alpha_{\mathrm{next}} \leftarrow \alpha_{\mathrm{current}} + \eta \, \nabla J(\alpha) \tag{17}$$

As with C in SVM, setting the learning rate η is a practiced science and art. A small η slows down learning, and a large η may lead to instability and divergence. In neural networks, researchers adapt the rate online as the training progresses [19]. Based on the experience of many WH users published on the Web, we tried values between 0.05 and 0.2 and settled for 0.1. This single, fixed value gave us results that were essentially as good as we could get by tuning η separately for each data set.

We must also fix our "convergence policy." In SIMPL, we repeat hill-climbing until the increase in $J(\alpha)$ is less than 5% over three successive iterations. This policy protects us from mild oscillations near the local optimum.

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 Sect. 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. In the unlikely situation that available RAM is smaller than $20m$ bytes, the expressions in Fig. 2 can be expressed as `GROUP BY` and aggregate operations, which can be executed efficiently with limited memory.

3.2 Pruning the training set

After a suitable number of hill-climbing steps, we discard points in D that are correctly classified 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 correctly classified documents is $O(n \log n)$ (and the total space needed is $O(m + n)$).

$i \leftarrow 0$
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
  remove from D those instances that are correctly classified by $\alpha^{(i)}$
  orthogonalize $\alpha^{(i)}$ w.r.t. $\alpha^{(0)}, \ldots, \alpha^{(i-1)}$ and scale it so that its L_2 norm $\|\alpha^{(i)}\| = 1$
  $i \leftarrow i + 1$
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

Fig. 1. The proposed algorithm SIMPL finds several linear discriminants $\alpha^{(0)}, \ldots, \alpha^{(k-1)}$ by using a hill-climbing procedure

$$J(\alpha) = \frac{\overbrace{\left(\frac{1}{|X|}\sum_X x\cdot\alpha - \frac{1}{|Y|}\sum_Y y\cdot\alpha\right)^2}^{N}}{\underbrace{\frac{1}{|X|}\sum_X (x\cdot\alpha)^2 - \left(\frac{1}{|X|}\sum_X x\cdot\alpha\right)^2}_{D_X} + \underbrace{\frac{1}{|Y|}\sum_Y (y\cdot\alpha)^2 - \left(\frac{1}{|Y|}\sum_Y y\cdot\alpha\right)^2}_{D_Y}} \tag{13}$$

$$\frac{\partial J}{\partial \alpha_i} = \frac{(D_X + D_Y)\frac{\partial N}{\partial \alpha_i} - N\frac{\partial (D_X + D_Y)}{\partial \alpha_i}}{(D_X + D_Y)^2} \tag{14}$$

$$\frac{\partial N}{\partial \alpha_i} = 2\left(\frac{1}{|X|}\sum_X x\cdot\alpha - \frac{1}{|Y|}\sum_Y y\cdot\alpha\right)\left(\frac{1}{|X|}\sum_X x_i - \frac{1}{|Y|}\sum_Y y_i\right) \tag{15}$$

$$\frac{\partial D_X}{\partial \alpha_i} = \frac{2}{|X|}\left(\sum_X x_i(x\cdot\alpha) - \frac{1}{|X|}\left(\sum_X x_i\right)\left(\sum_X x\cdot\alpha\right)\right) \tag{16}$$

Fig. 2. The main equations involved in the hill-climbing step to find the Fisher's linear discriminant

The intuition behind this algorithm is quite simple: having found a discriminant α we should retain only those points that fail to be separated in that direction. Although the hill-climbing steps will always converge to local optima, there is no bound on the number of αs we will need to extract. In practice, 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 documents in a new file, which then becomes our D for finding the next α.

Orthogonalizing the set of αs reduces the correlation between the components of the k-dimensional representation of documents. Because decision trees implement orthogonal cuts, we sometimes found a mild improvement in accuracy from the orthogonalization step; it never hurt accuracy. If we replace the decision tree by some other kind of learner, this step may not be needed.

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 [38] (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 very well for large m (number of dimensions), which is why decision trees induced on raw text show poor accuracy. This is also why our dimensionality reduction via projection pays off well.

C4.5 and WEKA hold the entire training data in memory, which is usually not a problem because by this stage we have transformed the training data to points with only two to three dimensions. For example, with 100000 documents and three projections, we will need only 1.2 MB. Should space become

an issue, we can always use efficient, out-of-core decision-tree implementations like SPRINT [34].

SIMPL induces fairly small decision trees. The number of projected coordinates is already small (two to three) to start with. It is rare to see more than two to three cuts on any one projected dimension. In our experiments, we observe that typically, the depth of the decision tree is less than five, and there are at most a total of 10–15 decision nodes in the tree.

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 Pentium III machines with 500–700 MHz CPUs and 512–2048 MB of RAM.

Several implementations of SVM are widely used and publicly available: Sequential Minimum Optimization (SMO) by John Platt [8,30], NodeLib by Gary Flake (`http://www.neci.nec.com/homepages/flake/nodelib/html/`), and SVMLIGHT by Thorsten Joachims [15]. We found SVMLIGHT to be comparable or better in accuracy compared to published SMO numbers. NodeLib has no built-in support for massive, sparse input data at this time. Our experiments are based on SVMLIGHT, evaluating it for C between 1 and 60 (SVMLIGHT's default is the reciprocal of the average value of $d \cdot d$, which range between 20 and 30 for our TFIDF representation). 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 $\bar{c}$	Estimated class 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 $P = n_{11}/(n_{11} + n_{01})$, respectively. $F_1 = 2RP/(R + P)$ 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."

4.1 Data sets

We use one synthetic data set and several standard real-life benchmark data sets. We use the synthetic data to study properties of the discriminants found by SVM and SIMPL, and we report precision and recall numbers on the real-life data sets.

4.1.1 Synthetic data

Our synthetic data generator is based on the "TCAT concept" defined by Joachims [16]. A TCAT concept is specified via a set of 3-tuples $\{(p_i, n_i, f_i), i = 1, 2, \ldots\}$. The ith tuple specifies a vocabulary subset having f_i terms. The overall vocabulary is the union of all these subsets. Positive documents ($c = 1$) use p_i out of the f_i terms from subset i, whereas negative documents ($c = -1$) use n_i of the f_i terms from subset i. Note that the same term may be picked for both positive and negative documents.

WebKB-course

p_i	n_i	f_i	Feature type
77	29	98	High-frequency positive (HFP)
4	21	52	High-frequency negative (HFN)
16	2	431	Medium-frequency positive (MFP)
1	12	341	Medium-frequency negative (MFN)
9	1	5045	Low-frequency positive (LFP)
1	21	24276	Low-frequency negative (LFN)
169	191	8116	Rest

Reuters-earn

p_i	n_i	f_i	Feature type
33	2	65	High frequency positive (HFP)
32	65	152	High frequency negative (HFN)
2	1	171	Medium frequency positive (MFP)
3	21	974	Medium frequency negative (MFN)
3	1	3455	Low frequency positive (LFP)
1	10	17020	Low frequency negative (LFN)
78	52	5821	Rest

Fig. 3. Parameters for the TCAT-based synthetic data generator

Joachims argued through examples that TCAT concepts closely model real-life text classification tasks. He estimated TCAT parameters for some well-known classification benchmarks (Sect. 4.1.2), two of which are shown in Fig. 3 and are used in our experiments.

We use a simple TCAT-based data generator, which chooses the p_i (respectively, n_i) terms uniformly at random, with replacement, from the f_i available features in set i. This implies all the synthetic documents have the same length (number of words), which is not very realistic. Note that our TCAT-based synthetic data are used purely to compare the linear discriminants computed by SVM and SIMPL under controlled circumstances, not to judge overall accuracy. Joachims used the TCAT characterization in a more powerful manner: he proved that TCAT concepts are learnable by LSVMs with small testing error, without making any assumptions about how the words are picked and how often they are repeated. He also derived results involving noise term distributions added to the basic TCAT model.

4.1.2 Real-life data

We use the following real-life 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 approach the scale we envisage for real applications. They were mainly used to compare run-time performance.

Reuters [21]: 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 and can be downloaded from `http://kdd.ics.uci.edu/databases/20newsgroups/20newsgroups.html`

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 and can be downloaded from `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-20/www/data/`.

OHSUMED: 348566 abstracts from medical journals, having around 230000 terms and 308511 topics and available at `http://ftp.ics.uci.edu/pub/machine-learning-databases/ohsumed/`. The raw text is 400 MB. The first 75% are selected as training documents and the rest are test documents.

Dmoz: From the RDF file published at `http://dmoz.org/`, we picked a sample of 140000 URLs, successfully crawled some 120000 of them, and stripped HTML tags, leaving plain text behind. The number of distinct tokens was over 500000. Twelve top-level topics of Dmoz were used; these were populated with 5000–21000 labeled documents each. The raw text (available on request) occupied 271 MB.

4.2 Document representation

We use the standard multinomial model (Eq. 2) for NB and the standard "TFIDF" document representation from IR for SVM and SIMPL. 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 $\mathrm{TF}(d,t) = 1 + \ln\big(1 + \ln(n(d,t))\big)$, 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 classifiers.

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 [25] (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.

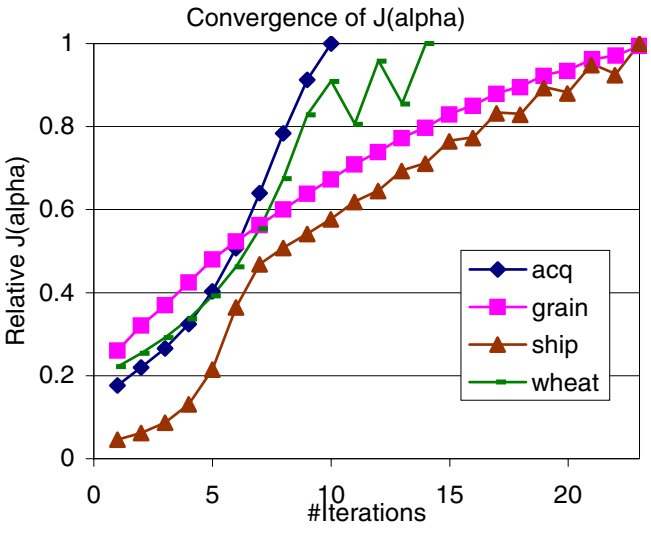

Fig. 4. Hill-climbing to maximize $J(\alpha)$ is fast. Relative values of $J(\alpha)$ (scaled to 1 at convergence) are plotted against the number of hill-climbing steps

4.3 Hill climbing

The hill-climbing approach is fast and practical. We usually settle at a maximum within 15–25 iterations: Fig. 4 shows that $J(\alpha)$ quickly grows and stabilizes with successive iterations. Our convergence policy (see Ssect. 3.1) guards well against mild problems of local maxima and overshoots in case the learning rate η in Eq. 17, which is set to 0.1 throughout, is not the best possible choice. 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.

An important concern for us was the lack of a guarantee of global optimality in the hill-climbing step. Suppose the hill-climbing process for $\alpha^{(0)}$ gets trapped in a local maximum, and we accept this value, continuing with $\alpha^{(1)}$ etc. Might this completely upset the optimization of subsequent αs and lead us arbitrarily astray from the best set of αs?

It is difficult to find a perfect answer to this question because an exhaustive search for all the optimal αs is infeasible, i.e., we have no access to "ground truth." Instead, to get some heuristic evidence of the robustness of SIMPL, we calculated two different values of $\alpha^{(0)}$:

- Perform hill climbing for $\alpha^{(0)}$ until convergence (this may give a global or a local optimum). Suppose this takes k iterations.
- Degrade the value of $\alpha^{(0)}$ deliberately by running the hill-climbing process for only $k/2$ iterations.