Scalable Nonlinear Learning with Adaptive Polynomial Expansions

Alekh Agarwal
Microsoft Research
alekha@microsoft.com

Alina Beygelzimer
Yahoo! Labs
beygel@yahoo-inc.com

Daniel Hsu
Columbia University
djhsu@cs.columbia.edu

John Langford
Microsoft Research
jcl@microsoft.com

Matus Telgarsky[*]
Rutgers University
mtelgars@cs.ucsd.edu

Abstract

Can we effectively learn a nonlinear representation in time comparable to linear learning? We describe a new algorithm that explicitly and adaptively expands higher-order interaction features over base linear representations. The algorithm is designed for extreme computational efficiency, and an extensive experimental study shows that its computation/prediction tradeoff ability compares very favorably against strong baselines.

1 Introduction

When faced with large datasets, it is commonly observed that using all the data with a simpler algorithm is superior to using a small fraction of the data with a more computationally intense but possibly more effective algorithm. The question becomes: What is the most sophisticated algorithm that can be executed given a computational constraint?

At the largest scales, Naïve Bayes approaches offer a simple, easily distributed single-pass algorithm. A more computationally difficult, but commonly better-performing approach is large scale linear regression, which has been effectively parallelized in several ways on real-world large scale datasets [1, 2]. Is there a modestly more computationally difficult approach that allows us to commonly achieve superior statistical performance?

The approach developed here starts with a fast parallelized online learning algorithm for linear models, and explicitly and adaptively adds higher-order interaction features over the course of training, using the learned weights as a guide. The resulting space of polynomial functions increases the approximation power over the base linear representation at a modest increase in computational cost.

Several natural folklore baselines exist. For example, it is common to enrich feature spaces with n-grams or low-order interactions. These approaches are naturally computationally appealing, because these nonlinear features can be computed on-the-fly avoiding I/O bottlenecks. With I/O bottlenecked datasets, this can sometimes even be done so efficiently that the additional computational complexity is negligible, so improving over this baseline is quite challenging.

The design of our algorithm is heavily influenced by considerations for computational efficiency, as discussed further in Section 2. Several alternative designs are plausible but fail to provide adequate computation/prediction tradeoffs or even outperform the aforementioned folklore baselines. An extensive experimental study in Section 3 compares efficient implementations of these baselines with

[*]This work was performed while MT was visiting Microsoft Research, NYC.

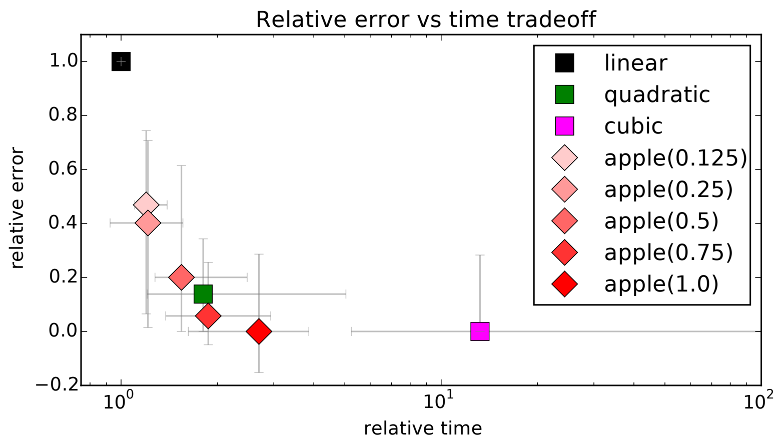


Figure 1: Computation/prediction tradeoff points using non-adaptive polynomial expansions and adaptive polynomial expansions (`apple`). The markers are positioned at the coordinate-wise median of (`relative error`, `relative time`) over 30 datasets, with bars extending to 25th and 75th percentiles. See Section 3 for definition of relative error and relative time used here.

the proposed mechanism and gives strong evidence of the latter's dominant computation/prediction tradeoff ability (see Figure 1 for an illustrative summary).

Although it is notoriously difficult to analyze nonlinear algorithms, it turns out that two aspects of this algorithm are amenable to analysis. First, we prove a regret bound showing that we can effectively compete with a growing feature set. Second, we exhibit simple problems where this algorithm is effective, and discuss a worst-case consistent variant. We point the reader to the full version [3] for more details.

Related work. This work considers methods for enabling nonlinear learning directly in a highly-scalable learning algorithm. Starting with a fast algorithm is desirable because it more naturally allows one to improve statistical power by spending more computational resources until a computational budget is exhausted. In contrast, many existing techniques start with a (comparably) slow method (e.g., kernel SVM [4], batch PCA [5], batch least-squares regression [5]), and speed it up by sacrificing statistical power, often just to allow the algorithm to run at all on massive data sets.

A standard alternative to explicit polynomial expansions is to employ polynomial kernels with the kernel trick [6]. While kernel methods generally have computation scaling at least quadratically with the number of training examples, a number of approximations schemes have been developed to enable a better tradeoff. The Nyström method (and related techniques) can be used to approximate the kernel matrix while permitting faster training [4]. However, these methods still suffer from the drawback that the model size after n examples is typically $O(n)$. As a result, even single pass online implementations [7] typically suffer from $O(n^2)$ training and $O(n)$ testing time complexity.

Another class of approximation schemes for kernel methods involves random embeddings into a high (but finite) dimensional Euclidean space such that the standard inner product there approximates the kernel function [8–11]. Recently, such schemes have been developed for polynomial kernels [9–11] with computational scaling roughly linear in the polynomial degree. However, for many sparse, high-dimensional datasets (such as text data), the embedding of [10] creates dense, high dimensional examples, which leads to a substantial increase in computational complexity. Moreover, neither of the embeddings from [9, 10] exhibits good statistical performance unless combined with dense linear dimension reduction [11], which again results in dense vector computations. Such feature construction schemes are also typically unsupervised, while the method proposed here makes use of label information.

Among methods proposed for efficiently learning polynomial functions [12–16], all but [13] are batch algorithms. The method of [13] uses online optimization together with an adaptive rule for creating interaction features. A variant of this is discussed in Section 2 and is used in the experimental study in Section 3 as a baseline.

Algorithm 1 Adaptive Polynomial Expansion (`apple`)

```
input  Initial features $S_1 = \{x_1, \ldots, x_d\}$, expansion sizes $(s_k)$, epoch schedule $(\tau_k)$, stepsizes $(\eta_t)$.
 1: Initial weights $\boldsymbol{w}_1 := \mathbf{0}$, initial epoch $k := 1$, parent set $P_1 := \emptyset$.
 2: for $t = 1, 2, \ldots$ : do
 3:     Receive stochastic gradient $\boldsymbol{g}_t$.
 4:     Update weights:   $\boldsymbol{w}_{t+1} := \boldsymbol{w}_t - \eta_t [\boldsymbol{g}_t]_{S_k}$,
        where $[\cdot]_{S_k}$ denotes restriction to monomials in the feature set $S_k$.
 5:     if $t = \tau_k$ then
 6:         Let $M_k \subseteq S_k$ be the top $s_k$ monomials $m(\boldsymbol{x}) \in S_k$ such that $m(\boldsymbol{x}) \notin P_k$, ordered from
            highest-to-lowest by the weight magnitude in $\boldsymbol{w}_{t+1}$.
 7:         Expand feature set:   $S_{k+1} := S_k \cup \{x_i \cdot m(\boldsymbol{x}) : i \in [d], m(\boldsymbol{x}) \in M_k\}$,   and
                                  $P_{k+1} := P_k \cup \{m(\boldsymbol{x}) : m(\boldsymbol{x}) \in M_k\}$.
 8:         $k := k + 1$.
 9:     end if
10: end for
```

2 Adaptive polynomial expansions

This section describes our new learning algorithm, `apple`.

2.1 Algorithm description

The pseudocode is given in Algorithm 1. The algorithm proceeds as stochastic gradient descent over the current feature set to update a weight vector. At specified times τ_k, the feature set S_k is expanded to S_{k+1} by taking the top monomials in the current feature set, ordered by weight magnitude in the current weight vector, and creating interaction features between these monomials and $\boldsymbol{x}$. Care is exercised to not repeatedly pick the same monomial for creating higher order monomial by tracking a parent set P_k, the set of all monomials for which higher degree terms have been expanded. We provide more intuition for our choice of this feature growing heuristic in Section 2.3.

There are two benefits to this staged process. Computationally, the stages allow us to amortize the cost of the adding of monomials—which is implemented as an expensive dense operation—over several other (possibly sparse) operations. Statistically, using stages guarantees that the monomials added in the previous stage have an opportunity to have their corresponding parameters converge. We have found it empirically effective to set $s_k := \text{average}\, \|[\boldsymbol{g}_t]_{S_1}\|_0$, and to update the feature set at a constant number of equally-spaced times over the entire course of learning. In this case, the number of updates (plus one) bounds the maximum degree of any monomial in the final feature set.

2.2 Shifting comparators and a regret bound for regularized objectives

Standard regret bounds compare the cumulative loss of an online learner to the cumulative loss of a *single* predictor (comparator) from a fixed comparison class. *Shifting regret* is a more general notion of regret, where the learner is compared to a *sequence* of comparators $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_T$.

Existing shifting regret bounds can be used to loosely justify the use of online gradient descent methods over expanding feature spaces [17]. These bounds are roughly of the form $\sum_{t=1}^{T} f_t(\boldsymbol{w}_t) - f_t(\boldsymbol{u}_t) \lesssim \sqrt{T \sum_{t<T} \|\boldsymbol{u}_t - \boldsymbol{u}_{t+1}\|}$, where $\boldsymbol{u}_t$ is allowed to use the same features available to $\boldsymbol{w}_t$, and f_t is the convex cost function in step t. This suggests a relatively high cost for a substantial total change in the comparator, and thus in the feature space. Given a budget, one could either do a liberal expansion a small number of times, or opt for including a small number of carefully chosen monomials more frequently. We have found that the computational cost of carefully picking a small number of high quality monomials is often quite high. With computational considerations at the forefront, we will prefer a more liberal but infrequent expansion. This also effectively exposes the learning algorithm to a large number of nonlinearities quickly, allowing their parameters to jointly converge between the stages.

It is natural to ask if better guarantees are possible under some structure on the learning problem. Here, we consider the stochastic setting (rather than the harsher adversarial setting of [17]), and further assume that our objective takes the form

$$f(\boldsymbol{w}) := \mathbb{E}[\ell(\langle \boldsymbol{w}, \boldsymbol{x}y\rangle)] + \lambda \|\boldsymbol{w}\|^2/2, \tag{1}$$

where the expectation is under the (unknown) data generating distribution D over $(\boldsymbol{x}, y) \in S \times \mathbb{R}$, and ℓ is some convex loss function on which suitable restrictions will be placed. Here S is such that $S_1 \subseteq S_2 \subseteq \ldots \subseteq S$, based on the largest degree monomials we intend to expand. We assume that in round t, we observe a stochastic gradient of the objective f, which is typically done by first sampling $(\boldsymbol{x}_t, y_t) \sim D$ and then evaluating the gradient of the regularized objective on this sample.

This setting has some interesting structural implications over the general setting of online learning with shifting comparators. First, the fixed objective f gives us a more direct way of tracking the change in comparator through $f(\boldsymbol{u}_t) - f(\boldsymbol{u}_{t+1})$, which might often be milder than $\|\boldsymbol{u}_t - \boldsymbol{u}_{t+1}\|$. In particular, if $\boldsymbol{u}_t = \arg\min_{\boldsymbol{u} \in S_k} f(\boldsymbol{u})$ in epoch k, for a nested subspace sequence S_k, then we immediately obtain $f(\boldsymbol{u}_{t+1}) \leq f(\boldsymbol{u}_t)$. Second, the strong convexity of the regularized objective enables the possibility of faster $O(1/T)$ rates than prior work [17]. Indeed, in this setting, we obtain the following stronger result. We use the shorthand $\mathbb{E}_t[\cdot]$ to denote the conditional expectation at time t, conditioning over the data from rounds $1, \ldots, t-1$.

Theorem 1. *Let a distribution over $(\boldsymbol{x}, y)$, twice differentiable convex loss ℓ with $\ell \geq 0$ and $\max\{\ell', \ell''\} \leq 1$, and a regularization parameter $\lambda > 0$ be given. Recall the definition* (1) *of the objective f. Let $(\boldsymbol{w}_t, \boldsymbol{g}_t)_{t \geq 1}$ be as specified by* `apple` *with step size $\eta_t := 1/(\lambda(t+1))$, where $\mathbb{E}_t([\boldsymbol{g}_t]_{S_{(t)}}) = [\nabla f(\boldsymbol{w}_t)]_{S_{(t)}}$ and $S_{(t)}$ is the support set corresponding to epoch k_t at time t in* `apple`*. Then for any comparator sequence $(\boldsymbol{u}_t)_{t=1}^{\infty}$ satisfying $\boldsymbol{u}_t \in S_{(t)}$, for any fixed $T \geq 1$,*

$$\mathbb{E}\left(f(\boldsymbol{w}_{T+1}) - \frac{\sum_{t=1}^{T}(t+2) f(\boldsymbol{u}_t)}{\sum_{t=1}^{T}(t+2)}\right) \leq \frac{1}{T+1}\left(\frac{(X^2+\lambda)(X+\lambda D)^2}{2\lambda^2}\right),$$

where $X \geq \max_t \|\boldsymbol{x}_t y_t\|$ and $D \geq \max_t \max\{\|\boldsymbol{w}_t\|, \|\boldsymbol{u}_t\|\}$.

Quite remarkably, the result exhibits no dependence on the cumulative shifting of the comparators unlike existing bounds [17]. This is the first result of this sort amongst shifting bounds to the best of our knowledge, and the only one that yields $1/T$ rates of convergence even with strong convexity. Of course, we limit ourselves to the stochastic setting, and prove expected regret guarantees on the final predictor $\boldsymbol{w}_T$ as opposed to a bound on $\sum_{t=1}^{T} f(\boldsymbol{w}_t)/T$. A curious distinction is our comparator, which is a weighted average of $f(\boldsymbol{u}_t)$ as opposed to the more standard uniform average. Recalling that $f(\boldsymbol{u}_{t+1}) \leq f(\boldsymbol{u}_t)$ in our setting, this is a *strictly harder benchmark* than an unweighted average and overemphasizes the later comparator terms which are based on larger support sets. Indeed, this is a nice compromise between competing against $\boldsymbol{u}_T$, which is the hardest yardstick, and $\boldsymbol{u}_1$, which is what a standard non-shifting analysis compares to. Indeed our improvement can be partially attributed to the stability of the averaged f values as opposed to just $f(u_T)$ (more details in [3]). Overall, this result demonstrates that in our setting, while there is generally a cost to be paid for shifting the comparator too much, it can still be effectively controlled in favorable cases. One problem for future work is to establish these fast $1/T$ rates also with high probability.

Note that the regret bound offers no guidance on how or when to select new monomials to add.

2.3 Feature expansion heuristics

Previous work on learning sparse polynomials [13] suggests that it is possible to anticipate the utility of interaction features before even evaluating them. For instance, one of the algorithms from [13] orders monomials $m(\boldsymbol{x})$ by an estimate of $\mathbb{E}[r(\boldsymbol{x})^2 m(\boldsymbol{x})^2]/\mathbb{E}[m(\boldsymbol{x})^2]$, where $r(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}] - \hat{f}(\boldsymbol{x})$ is the residual of the current predictor $\hat{f}$ (for least-squares prediction of the label y). Such an index is shown to be related to the potential error reduction by polynomials with $m(\boldsymbol{x})$ as a factor. We call this the SSM heuristic (after the authors of [13], though it differs from their original algorithm).

Another plausible heuristic, which we use in Algorithm 1, simply orders the monomials in S_k by their weight magnitude in the current weight vector. We can justify this weight heuristic in the following simple example. Suppose a target function $\mathbb{E}[y|\boldsymbol{x}]$ is just a single monomial in $\boldsymbol{x}$, say, $m(\boldsymbol{x}) := \prod_{i \in M} x_i$ for some $M \subseteq [d]$, and that $\boldsymbol{x}$ has a product distribution over $\{0,1\}^d$ with $0 < \mathbb{E}[x_i] =: p \leq 1/2$ for all $i \in [d]$. Suppose we repeatedly perform 1-sparse regression with the current feature set S_k, and pick the top weight magnitude monomial for inclusion in the parent set P_{k+1}. It is easy to show that the weight on a degree ℓ sub-monomial of $m(\boldsymbol{x})$ in this regression is $p^{|M|-\ell}$, and the weight is strictly smaller for any term which is not a proper sub-monomial of $m(\boldsymbol{x})$. Thus we repeatedly pick the largest available sub-monomial of $m(\boldsymbol{x})$ and expand it, eventually discovering $m(\boldsymbol{x})$. After k stages of the algorithm, we have at most kd features in our regression here, and hence we find $m(\boldsymbol{x})$ with a total of $d|M|$ variables in our regression, as opposed to $d^{|M|}$ which typical feature selection approaches would need. This intuition can be extended more generally to scenarios where we do not necessarily do a sparse regression and beyond product distributions, but we find that even this simplest example illustrates the basic motivations underlying our choice—we want to parsimoniously expand on top of a base feature set, while still making progress towards a good polynomial for our data.

2.4 Fall-back risk-consistency

Neither the SSM heuristic nor the weight heuristic is rigorously analyzed (in any generality). Despite this, the basic algorithm `apple` can be easily modified to guarantee a form of risk consistency, regardless of which feature expansion heuristic is used. Consider the following variant of the support update rule in the algorithm `apple`. Given the current feature budget s_k, we add $s_k - 1$ monomials ordered by weight magnitudes as in Step 7. We also pick a monomial $m(\boldsymbol{x})$ of the smallest degree such that $m(\boldsymbol{x}) \notin P_k$. Intuitively, this ensures that all degree 1 terms are in P_k after d stages, all degree 2 terms are in P_k after $k = O(d^2)$ stages and so on. In general, it is easily seen that $k = O(d^{\ell-1})$ ensures that all degree $\ell - 1$ monomials are in P_k and hence all degree ℓ monomials are in S_k. For ease of exposition, let us assume that s_k is set to be a constant s independent of k. Then the total number of monomials in P_k when $k = O(d^{\ell-1})$ is $O(sd^{\ell-1})$, which means the total number of features in S_k is $O(sd^\ell)$.

Suppose we were interested in competing with all γ-sparse polynomials of degree ℓ. The most direct approach would be to consider the explicit enumeration of all monomials of degree up to ℓ, and then perform ℓ_1-regularized regression [18] or a greedy variable selection method such as OMP [19] as means of enforcing sparsity. This ensures consistent estimation with $n = O(\gamma \log d^\ell) = O(\gamma \ell \log d)$ examples. In contrast, we might need $n = O(\gamma(\ell \log d + \log s))$ examples in the worst case using this fall back rule, a minor overhead at best. However, in favorable cases, we stand to gain a lot when the heuristic succeeds in finding good monomials rapidly. Since this is really an empirical question, we will address it with our empirical evaluation.

3 Experimental study

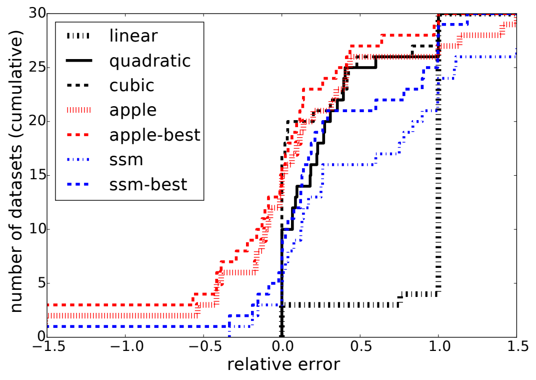
We now describe of our empirical evaluation of `apple`.

3.1 Implementation, experimental setup, and performance metrics

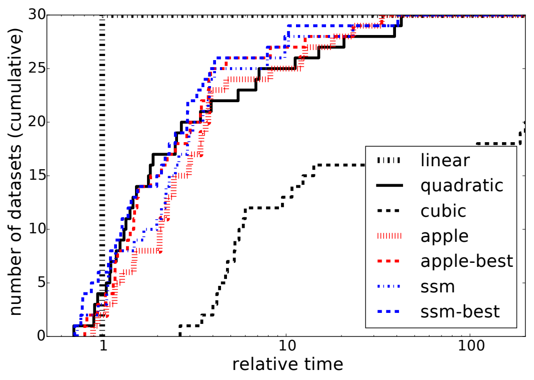
In order to assess the effectiveness of our algorithm, it is critical to build on top of an efficient learning framework that can handle large, high-dimensional datasets. To this end, we implemented `apple` in the Vowpal Wabbit (henceforth VW) open source machine learning software[1]. VW is a good framework for us, since it also natively supports quadratic and cubic expansions on top of the base features. These expansions are done dynamically at run-time, rather than being stored and read from disk in the expanded form for computational considerations. To deal with these dynamically enumerated features, VW uses hashing to associate features with indices, mapping each feature to a b-bit index, where b is a parameter. The core learning algorithm is an online algorithm as assumed in `apple`, but uses refinements of the basic stochastic gradient descent update (e.g., [20–23]).

We implemented `apple` such that the total number of epochs was always 6 (meaning 5 rounds of adding new features). At the end of each epoch, the non-parent monomials with largest magnitude weights were marked as parents. Recall that the number of parents is modulated at s^α for some $\alpha > 0$, with s being the average number of non-zero features per example in the dataset so far. We will present experimental results with different choices of α, and we found $\alpha = 1$ to be a reliable

[1]Please see `https://github.com/JohnLangford/vowpal_wabbit` and the associated git repository, where `-stage_poly` and related command line options execute `apple`.



(a)



(b)

Figure 2: Dataset CDFs across all 30 datasets: (a) relative test error, (b) relative training time (log scale). $\{\texttt{apple}, \texttt{ssm}\}$ refer to the $\alpha = 1$ default; $\{\texttt{apple}, \texttt{ssm}\}$-best picks best α per dataset.

default. Upon seeing an example, the features are enumerated on-the-fly by recursively expanding the marked parents, taking products with base monomials. These operations are done in a way to respect the sparsity (in terms of base features) of examples which many of our datasets exhibit.

Since the benefits of nonlinear learning over linear learning themselves are very dataset dependent, and furthermore can vary greatly for different heuristics based on the problem at hand, we found it important to experiment with a large testbed consisting of a diverse collection of medium and large-scale datasets. To this end, we compiled a collection of 30 publicly available datasets, across a number of KDDCup challenges, UCI repository and other common resources (detailed in the appendix). For all the datasets, we tuned the learning rate for each learning algorithm based on the progressive validation error (which is typically a reliable bound on test error) [24]. The number of bits in hashing was set to 18 for all algorithms, apart from cubic polynomials, where using 24 bits for hashing was found to be important for good statistical performance. For each dataset, we performed a random split with 80% of the data used for training and the remainder for testing. For all datasets, we used squared-loss to train, and 0-1/squared-loss for evaluation in classification/regression problems. We also experimented with ℓ_1 and ℓ_2 regularization, but these did not help much. The remaining settings were left to their VW defaults.

For aggregating performance across 30 diverse datasets, it was important to use error and running time measures on a scale independent of the dataset. Let ℓ, $\texttt{q}$ and $\texttt{c}$ refer to the test errors of linear, quadratic and cubic baselines respectively (with `lin`, `quad`, and `cubic` used to denote the baseline algorithms themselves). For an algorithm `alg`, we compute the *relative (test) error*:

$$\mathrm{rel\,err}(\texttt{alg}) = \frac{\mathrm{err}(\texttt{alg}) - \min(\ell, \texttt{q}, \texttt{c})}{\max(\ell, \texttt{q}, \texttt{c}) - \min(\ell, \texttt{q}, \texttt{c})}, \tag{2}$$

where $\min(\ell, \texttt{q}, \texttt{c})$ is the smallest error among the three baselines on the dataset, and $\max(\ell, \texttt{q}, \texttt{c})$ is similarly defined. We also define the *relative (training) time* as the ratio to running time of `lin`: $\mathrm{rel\,time}(\texttt{alg}) = \mathrm{time}(\texttt{alg})/\mathrm{time}(\texttt{lin})$. With these definitions, the aggregated plots of relative errors and relative times for the various baselines and our methods are shown in Figure 2. For each method, the plots show a cumulative distribution function (CDF) across datasets: an entry (a, b) on the left plot indicates that the relative error for b datasets was at most a. The plots include the baselines `lin`, `quad`, `cubic`, as well as a variant of `apple` (called `ssm`) that replaces the weight heuristic with the SSM heuristic, as described in Section 2.3. For `apple` and `ssm`, the plot shows the results with the fixed setting of $\alpha = 1$, as well as the best setting chosen per dataset from $\alpha \in \{0.125, 0.25, 0.5, 0.75, 1\}$ (referred to as `apple`-best and `ssm`-best).

3.2 Results

In this section, we present some aggregate results. Detailed results with full plots and tables are presented in the appendix. In the Figure 2(a), the relative error for all of `lin`, `quad` and `cubic` is

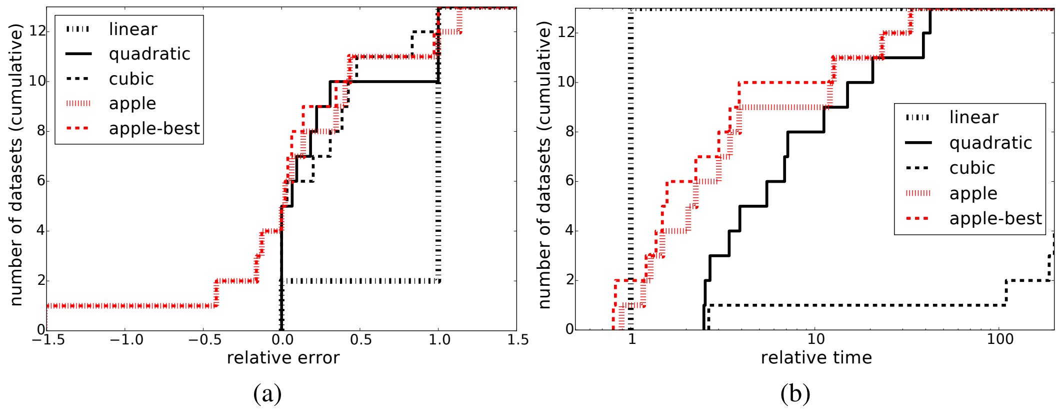


Figure 3: Dataset CDFs across 13 datasets where $\mathrm{time}(\texttt{quad}) \geq 2\mathrm{time}(\texttt{lin})$: (a) relative test error, (b) relative training time (log scale).

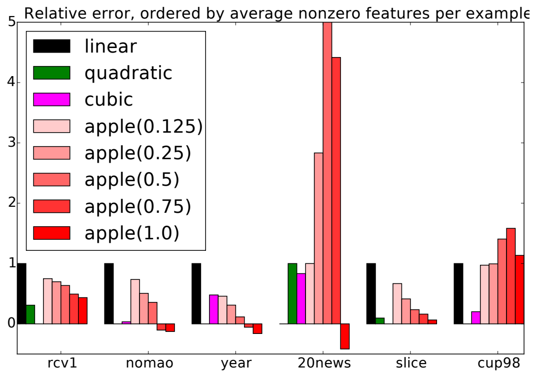
always to the right of 0 (due to the definition of $\mathrm{rel\,err}$). In this plot, a curve enclosing a larger area indicates, in some sense, that one method uniformly dominates another. Since `apple` uniformly dominates `ssm` statistically (with only slightly longer running times), we restrict the remainder of our study to comparing `apple` to the baselines `lin`, `quad` and `cubic`. We found that on 12 of the 30 datasets, the relative error was negative, meaning that `apple` beats all the baselines. A relative error of 0.5 indicates that we cover at least half the gap between $\min(\ell, \mathtt{q}, \mathtt{c})$ and $\max(\ell, \mathtt{q}, \mathtt{c})$. We find that we are below 0.5 on 27 out of 30 datasets for `apple`-best, and 26 out of the 30 datasets for the setting $\alpha = 1$. This is particularly striking since the error $\min(\ell, \mathtt{q}, \mathtt{c})$ is attained by `cubic` on a majority of the datasets (17 out of 30), where the relative error of `cubic` is 0. Hence, statistically `apple` often outperforms even `cubic`, while typically using a much smaller number of features. To support this claim, we include in the appendix a plot of the average number of features per example generated by each method, for all datasets. Overall, we find the statistical performance of `apple` from Figure 2 to be quite encouraging across this large collection of diverse datasets.

The running time performance of `apple` is also extremely good. Figure 2(b) shows that the running time of `apple` is within a factor of 10 of `lin` for almost all datasets, which is quite impressive considering that we generate a potentially much larger number of features. The gap between `lin` and `apple` is particularly small for several large datasets, where the examples are sparse and high-dimensional. In these cases, all algorithms are typically I/O-bottlenecked, which is the same for all algorithms due to the dynamic feature expansions used. It is easily seen that the statistically efficient baseline of `cubic` is typically computationally infeasible, with the relative time often being as large as 10^2 and 10^5 on the biggest dataset. Overall, the statistical performance of `apple` is competitive with and often better than $\min(\ell, \mathtt{q}, \mathtt{c})$, and offers a nice intermediate in computational complexity.

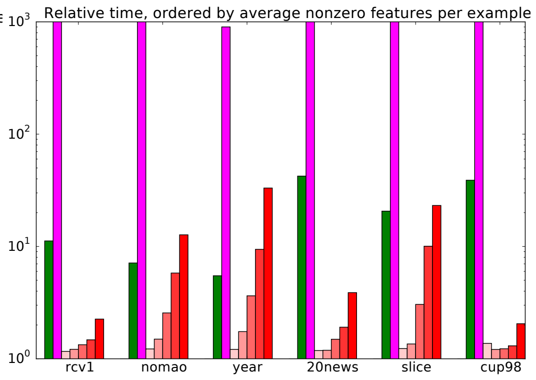
A surprise in Figure 2(b) is that `quad` appears to computationally outperform `apple` for a relatively large number of datasets, at least in aggregate. This is due to the extremely efficient implementation of `quad` in VW: on 17 of 30 datasets, the running time of `quad` is less than twice that of `lin`. While we often statistically outperform `quad` on many of these smaller datasets, we are primarily interested in the larger datasets where the relative cost of nonlinear expansions (as in `quad`) is high.

In Figure 3, we restrict attention to the 13 datasets where $\mathrm{time}(\texttt{quad})/\mathrm{time}(\texttt{lin}) \geq 2$. On these larger datasets, our statistical performance seems to dominate all the baselines (at least in terms of the CDFs, more on individual datasets will be said later). In terms of computational time, we see that we are often much better than `quad`, and `cubic` is essentially infeasible on most of these datasets. This demonstrates our key intuition that such adaptively chosen monomials are key to effective nonlinear learning in large, high-dimensional datasets.

We also experimented with *picky* algorithms of the sort mentioned in Section 2.2. We tried the original algorithm from [13], which tests a candidate monomial before adding it to the feature set S_k, rather than just testing candidate parent monomials for inclusion in P_k; and also a picky algorithm based on our weight heuristic. Both algorithms were extremely computationally expensive, even when implemented using VW as a base: the explicit testing for inclusion in S_k (on a per-example



(a)



(b)

Figure 4: Comparison of different methods on the top 6 datasets by non-zero features per example: (a) relative test errors, (b) relative training times.

	`lin`	`lin` + `apple`	`bigram`	`bigram` + `apple`
Test AUC	0.81664	0.81712	0.81757	0.81796
Training time (in s)	1282	2727	2755	7378

Table 1: Test error and training times for different methods in a large-scale distributed setting. For $\{\texttt{lin}, \texttt{bigram}\} + \texttt{apple}$, we used $\alpha = 0.25$.

basis) caused too much overhead. We ruled out other baselines such as polynomial kernels for similar computational reasons.

To provide more intuition, we also show individual results for the top 6 datasets with the highest average number of non-zero features per example—a key factor determining the computational cost of all approaches. In Figure 4, we show the performance of the `lin`, `quad`, `cubic` baselines, as well as `apple` with 5 different parameter settings in terms of relative error (Figure 4(a)) and relative time (Figure 4(b)). The results are overall quite positive. We see that on 3 of the datasets, we improve upon all the baselines statistically, and even on other 3 the performance is quite close to the best of the baselines with the exception of the `cup98` dataset. In terms of running time, we find `cubic` to be extremely expensive in all the cases. We are typically faster than `quad`, and in the few cases where we take longer, we also obtain a statistical improvement for the slight increase in computational cost. In conclusion, on larger datasets, the performance of our method is quite desirable.

Finally, we also implemented a parallel version of our algorithm, building on the repeated averaging approach [2, 25], using the built-in AllReduce communication mechanism of VW, and ran an experiment using an internal advertising dataset consisting of approximately 690M training examples, with roughly 318 non-zero features per example. The task is the prediction of `click/no-click` events. The data was stored in a large Hadoop cluster, split over 100 partitions. We implemented the `lin` baseline, using 5 passes of online learning with repeated averaging on this dataset, but could not run full `quad` or `cubic` baselines due to the prohibitive computational cost. As an intermediate, we generated `bigram` features, which only doubles the number of non-zero features per example. We parallelized `apple` as follows. In the first pass over the data, each one of the 100 nodes locally selects the promising features over 6 epochs, as in our single-machine setting. We then take the union of all the parents locally found across all nodes, and freeze that to be the parent set for the rest of training. The remaining 4 passes are now done with this fixed feature set, repeatedly averaging local weights. We then ran `apple`, on top of both `lin` as well as `bigram` as the base features to obtain maximally expressive features. The test error was measured in terms of the area under ROC curve (AUC), since this is a highly imbalanced dataset. The error and time results, reported in Table 1, show that using nonlinear features does lead to non-trivial improvements in AUC, albeit at an increased computational cost. Once again, this should be put in perspective with the full `quad` baseline, which did not finish in over a day on this dataset.

Acknowledgements: We thank Leon Bottou, Rob Schapire and Dean Foster for helpful discussions.

References

[1] I. Mukherjee, K. Canini, R. Frongillo, and Y. Singer. Parallel boosting with momentum. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2013.

[2] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15(Mar):1111–1133, 2014.

[3] A. Agarwal, A. Beygelzimer, D. Hsu, J. Langford, and M. Telgarsky. Scalable nonlinear learning with adaptive polynomial expansions. 2014. `arXiv:1410.0440 [cs.LG]`.

[4] C. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems 13*, 2001.

[5] M. W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends in Machine Learning*, 3(2):123–224, 2011.

[6] B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[7] A. Bordes, S. Ertekin, J. Weston, and L. Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579–1619, 2005.

[8] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems 20*, 2008.

[9] P. Kar and H. Karnick. Random feature maps for dot product kernels. In *AISTATS*, 2012.

[10] N. Pham and R. Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[11] R. Hamid, A. Gittens, Y. Xiao, and D. Decoste. Compact random feature maps. In *ICML*, 2014.

[12] A. G. Ivakhnenko. Polynomial theory of complex systems. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-1(4):364–378, 1971.

[13] T. D. Sanger, R. S. Sutton, and C. J. Matheus. Iterative construction of sparse polynomial approximations. In *Advances in Neural Information Processing Systems 4*, 1992.

[14] A. T. Kalai, A. Samorodnitsky, and S.-H. Teng. Learning and smoothed analysis. In *FOCS*, 2009.

[15] A. Andoni, R. Panigrahy, G. Valiant, and L. Zhang. Learning sparse polynomial functions. In *SODA*, 2014.

[16] A. G. Dimakis, A. Klivans, M. Kocaoglu, and K. Shanmugam. A smoothed analysis for learning sparse polynomials. *CoRR*, abs/1402.3902, 2014.

[17] M. Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *ICML*, 2003.

[18] R. Tibshirani. Regression shrinkage and selection via the lasso. *J. Royal. Statist. Soc B.*, 58(1):267–288, 1996.

[19] J. A. Tropp and A. C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655–4666, December 2007.

[20] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[21] H. B. McMahan and M. J. Streeter. Adaptive bound optimization for online convex optimization. In *COLT*, pages 244–256, 2010.

[22] N. Karampatziakis and J. Langford. Online importance weight aware updates. In *UAI*, pages 392–399, 2011.

[23] S. Ross, P. Mineiro, and J. Langford. Normalized online learning. In *UAI*, 2013.

[24] A. Blum, A. Kalai, and J. Langford. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *COLT*, 1999.

[25] K. Hall, S. Gilpin, and G. Mann. Mapreduce/bigtable for distributed optimization. In *Workshop on Learning on Cores, Clusters, and Clouds*, 2010.

[26] S. Bubeck. Theory of convex optimization for machine learning. 2014. `arXiv:1405.4980 [math.OC]`.

[27] O. Shamir and T. Zhang. Stochastic gradient descent for non-smooth optimization: Convergence results and optimal averaging schemes. In *ICML*, 2013.