Classifier Cascade for Minimizing Feature Evaluation Cost

Minmin Chen[1] **Zhixiang (Eddie) Xu**[1] **Kilian Q. Weinberger**[1] **Olivier Chapelle**[2] **Dor Kedem**[1]
Washington University in Saint Louis[1]
Saint Louis, MO
chenm, zhixiang.xu, kilian, kedem.dor@wustl.edu
Yahoo! Research[2]
Santa Clara, CA
chap@yahoo-inc.com

Abstract

Machine learning algorithms are increasingly used in large-scale industrial settings. Here, the operational cost during test-time has to be taken into account when an algorithm is designed. This operational cost is affected by the average running time and the computation time required for feature extraction. When a diverse set of features is used, the latter can vary drastically. In this paper we propose an algorithm that constructs a cascade of classifiers which explicitly trades-off operational cost and classifier accuracy while accounting for on-demand feature extraction costs. Different from previous work, our algorithm re-optimizes trained classifiers and allows expensive features to be scheduled at any stage within the cascade to minimize overall cost. Experiments on actual web-search ranking data sets demonstrate that our framework leads to drastic test-time improvements.

1 Introduction

During the past decade, the field of machine learning has managed a successful transition from academic research to industrial real-world applications. Silently, machine learning algorithms have entered the mainstream market through applications such as web-search engines (Zheng et al., 2008), product recommendations (Bennett and Lanning, 2007), family-safe content filtering (Fleck et al., 1996), email- and web-spam filters (Abernethy et al., 2008; Weinberger et al., 2009) and many others. These successes are a testament to the maturity of machine learning as a research field and to the robustness of the various algorithms and approaches. In web-search engines, machine learned rankers are used hundreds of millions of times per day and are relied upon by billions of people around the world.

However, there is a distinct difference between the machine learning scenarios in typical research papers and the real-world industrial setting. In industrial settings, the average computational cost during test-time is a serious consideration when algorithms are deployed. If a task is performed millions of times per day, it is crucial that the average computation time required per instance is sufficiently low to stay within the limits of the available computational resources. As an example consider a commercial e-mail spam filter. It has to process millions of messages per day, and given its limited resources must spend less than 10 milliseconds on each individual e-mail. Similarly, a web search engine might have to score hundreds of thousands of documents within a few milliseconds.

The two key differences from traditional machine learning are that 1. the computational cost is evaluated *on average* per *test instance* and 2. features are computed on-demand and vary significantly in cost. For example, in the e-mail spam filtering scenario, it is fine to spend 30 milliseconds on the occasional image spam e-mail (which might require expensive OCR features), if a majority of the messages can be filtered out just based on their sender-IP address in less than one millisecond (without even loading the message content). Similarly, in web search ranking, many documents can cheaply be eliminated using precomputed features such as PageRank. This leaves few documents to be scored with more expensive features such as advanced text match ones (Chapelle and Chang, 2011, Section 4.2).

Although the problem of fast evaluation has already been studied, mostly in the context of face detection (Viola and Jones, 2002), average-time complexity and on-demand feature-cost amortization is surprisingly absent in most of today's machine-learning literature – even in application domains such as web-search ranking (Chapelle et al., 2010; Zheng et al., 2008) or email-spam filtering (Weinberger et al., 2009).

In this paper we consider the problem of post-processing classifiers to reduce their test time complexity, based on

Appearing in Proceedings of the 15^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume XX of JMLR: W&CP XX.

classifier execution time and feature extraction cost in learning scenarios with skewed class proportions. In particular, we focus on re-weighting and re-ordering of weak learners obtained with additive regression methods (such as gradient boosting (Friedman, 2001)). After the initial training, a dictionary of classifiers is re-organized into a chain of cascades (Viola and Jones, 2002) – where each can reject an input or pass it on to the subsequent stage. The average cost is significantly reduced, because most test inputs are rejected early-on after only a few cheap features are extracted. Different from previous work (Raykar et al., 2010), we do not pre-assign features to cascade stages; rather, we make the order of the feature extraction part of the optimization. This entails a crucial benefit, as it allows even an expensive feature to be used early if it leads to an amortized gain in performance through more accurate and aggressive rejection of inputs.

Following several intuitive steps we derive an optimization problem which can be solved in a simple iterative procedure. We further relax the optimization problem into a series of sub-problems with simple closed-form solutions, which can be solved in a matter of minutes. We use this relaxed version as initialization and as an efficient method to cross-validate hyper-parameters. We demonstrate that our method reduces the average cost of a classifier during test time by an order of magnitude on real-world web-search ranking data sets, with only a marginal reduction in retrieval precision. Surprisingly, even our initialization already significantly outperforms the current state-of-the-art.

2 Related Work

Previous research has addressed the task of learning cascades of classifiers in various ways in the context of diverse applications. The most prominent related work is by Viola and Jones (2002) in the context of real-time face-recognition, which arguably inspired most related subsequent research. Their algorithm uses Adaboost (Schapire, 1999) to greedily train a cascade of classifiers. Similar to our approach, each stage can prematurely reject inputs or pass them on for further evaluation. The average test-time cost is reduced because early stages, which are applied to the majority of the inputs, are short and inexpensive. Different from our method, they do not incorporate individualized feature costs and have no global optimization across cascade stages. In the context of information retrieval, Wang et al. (2011) adapted this approach to ranking and incorporated feature costs but retained the underlying greedy paradigm. Cambazoglu et al. (2010) propose an approach to build on top of gradient boosted regression trees explicitly for web-search ranking. Similar to our method, they post-process an additive classifier into a cascade. However, in their setting the order and weights of the trees are fixed and they merely introduce early-exits into a fixed additive model. In contrast, our method re-arranges and re-weights the trees with the help of a global optimization problem.

Lefakis and Fleuret (2010) use soft-cascades to boost the entire cascade structure. Here, during training, at each stage inputs are weighted by their probability of passing all previous stages. The authors use a fixed number of stages and all cascades are created together by iteratively adding a weak classifier to each stage, maximizing the log-likelihood for the entire cascade model. Saberian and Vasconcelos (2010) propose a similar approach that takes into account both the classification loss and the generated cost. There is no set number of stages and new stages are generated during training. Each iteration, weak classifiers are built for every stage and only the one which optimizes the objective function the most is picked in a greedy fashion.

Most similar to our method is the work by Raykar et al. (2010). Here, the authors suggest to first group the weak classifiers and features according to their features' costs. They maximize the log-likelihood of the training data while minimizing the expected test-time cost. However, different from our method, each stage is constrained to only contain a fixed subset of features or weak-learners, whereas our method automatically assigns features and weak-learners globally.

3 Method

We assume that training data is provided in a set of (input, label) pairs $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_n, y_n)\} \in \mathcal{R}^d \times \{+1, -1\}$. For simplicity, the class labels are binary, although other settings are possible.

In our setting we assume that the two classes are highly unbalanced, in particular that there are many more *negative* examples than *positive* ones. This setting occurs in many real-world applications such as web-search ranking (almost all web-pages are of no relevance to a search-query), email-spam filtering (most email is spam) and many others (Lefakis and Fleuret, 2010; Raykar et al., 2010; Saberian and Vasconcelos, 2010; Viola and Jones, 2002).

We assume that we have a set of base hypotheses $\mathcal{H} = \{h_1, h_2, \cdots, h_T\}$ available. Each h_t could be a limited depth regression tree (Breiman et al., 1984) and the set $\mathcal{H}$ might have been obtained using gradient boosted decision trees (Friedman, 2001). In this paper we describe how to re-weight and re-arrange these weak learners into cascades of more cost-efficient classifier $f(\cdot) = \sum_t \beta_t h_t(\cdot)$.

We assume that each feature α has a specific acquisition cost $c_\alpha > 0$ with $\mathbf{c} \in \mathcal{R}^d_+$, which is required for its initial computation. Once a feature has been used for the first time, subsequent evaluations are free[1] as its value can be

[1]This second-use cost could also be set to a small positive constant without any major change in our problem formulation — for simplicity we assume it is zero.

cached efficiently. In addition to a feature cost we also assume a tree-evaluation cost, denoted by the vector $\mathbf{e} \in \mathcal{R}_+^T$, where e_t is the computational cost of evaluating tree t. (The vector $\mathbf{e}$ is all-constant if all trees are of identical depth.)

For notational simplicity, we define a binary matrix $\mathbf{F} \in \{0,1\}^{d\times T}$, where $F_{\alpha t} = 1$ if and only if feature α is used in tree t. We can express the cost of a particular classifier $f(\cdot) = \sum_t \beta_t h_t(\cdot)$ as

$$c(f) = \mathbf{e}^\top \delta(\boldsymbol{\beta}) + \mathbf{c}^\top \delta\left(\mathbf{F}\delta(\boldsymbol{\beta})\right), \tag{1}$$

where the function $\delta(a) \in \{0,1\}$ is defined as $\delta(a) = 0$ if and only if $a = 0$.[2] The first term within the sum in (1) is the accumulative tree-evaluation cost, while the second term sums over the feature costs of all the features that are used in at least one tree in f.

Equation (1) is the average cost for a single function f evaluating all samples; this is the single-stage setting of the next section. After that we will analyze a cascade model in which there is a different function at every stage.

3.1 Single-stage Optimization

As a first approach, given a set of trees $\mathcal{H}$ previously learned using gradient boosted regression trees or by any other method, we propose to *re-weight* these trees as to approximately minimize the following loss function $\mathcal{L}$ which trades off between accuracy and cost:

$$\mathcal{L}(f) = \ell(f) + \rho r(f) + \lambda c(f). \tag{2}$$

where $\ell(\cdot)$ is some convex regression loss function. The cost-term is weighted by $\lambda > 0$. To avoid over fitting, we also introduce an additional regularization term $r(f)$, weighted by $\rho > 0$.

For convenience, throughout the paper we use the weighted squared loss $\ell(f) = \frac{1}{2}\sum_{i=1}^n \omega_i (f(\mathbf{x}_i) - y_i)^2$, where ω_i is the weight pre-assigned to the loss of an input $\mathbf{x}_i$.[3] For balanced data usually $\omega_i = \omega = \frac{1}{n}$, but for unbalanced data, it is a common practice to weigh positive and negative classes differently, i.e., $\{\omega_i = \omega^+, \forall i \in C^+\}$ and $\{\omega_i = \omega^-, \forall i \in C^-\}$ where C^+ and C^- are the corresponding sets of indices for the positive and negative classes respectively.

For a given input $\mathbf{x}_i$, we create a vector $\mathbf{h}(\mathbf{x}_i) = [h_1(\mathbf{x}_i), \cdots, h_T(\mathbf{x}_i)]^\top$ that contains the regression results using all the trees in $\mathcal{H}$ (and a constant 1, which we assume is an element of $\mathcal{H}$). With this notation, we can express the resulting classifier as

$$f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{h}(\mathbf{x}). \tag{3}$$

We choose the weighted squared loss for $\ell(\cdot)$ and an ℓ_1 regularizer for $r(\cdot)$, and rewrite the loss function (2) entirely in terms of the weight vector $\boldsymbol{\beta}$: [4]

$$\mathcal{L}(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^n \omega_i (\boldsymbol{\beta}^\top \mathbf{h}(\mathbf{x}_i) - y_i)^2 + \rho\|\boldsymbol{\beta}\|_1 + \lambda\left[\mathbf{e}^\top \delta(\boldsymbol{\beta}) + \mathbf{c}^\top \delta(\mathbf{F}\delta(\boldsymbol{\beta}))\right]. \tag{4}$$

Our goal in this section is to optimize over the weight-vector $\boldsymbol{\beta}$ to minimize the cost-sensitive loss $\mathcal{L}(\cdot)$ as stated in eq.(4).

Relaxation. The loss in eq. (4) is non-continuous, but we can relax it to make it better behaved.

Tree Sparsity. The term $\mathbf{e}^\top\delta(\boldsymbol{\beta})$ in eq. (4) computes the accumulative tree evaluation cost. To achieve continuity, we relax the $\delta(\cdot)$ function into an absolute-value, resulting in the following relaxation:

$$\sum_{t=1}^T e_t \delta(\beta_t) \longrightarrow \sum_{t=1}^T e_t |\beta_t| \tag{5}$$

The resulting weighted-ℓ_1 regularization encourages sparsity in the weights and therefore fewer trees to be evaluated during test-time.

Feature Sparsity. The term $\mathbf{c}^\top\delta(\mathbf{F}\delta(\boldsymbol{\beta}))$ in eq. (4) calculates the total feature acquisitions cost. Each entry of the vector $\mathbf{F}\delta(\boldsymbol{\beta})$ denotes the count of active trees (*i.e.* $\beta_t \neq 0$) in which a particular feature is used. Here, we have two non-continuous $\delta(\cdot)$ functions. We relax both jointly into a weighted $\ell_1 - \ell_2$ mixed-norm (Kowalski, 2009),

$$\sum_{\alpha=1}^d c_\alpha \sum_{t=1}^T \delta(F_{\alpha t}\delta(\beta_t)) \longrightarrow \sum_{\alpha=1}^d c_\alpha \sqrt{\sum_{t=1}^T (F_{\alpha t}\beta_t)^2}. \tag{6}$$

Intuitively, the mixed-norm in (6) enforces sparsity on a per-feature level – effectively, for a given feature, forcing the classifier to pick many weak-learners that require it, or none at all. This is encouraged more heavily when a feature is expensive to compute for the first time (*i.e.* c_α is large).

Applying the two relaxations (5) and (6) to our objective (4), we obtain a continuous and convex objective function,

$$\hat{\mathcal{L}}(\boldsymbol{\beta}) = \frac{1}{2}\sum_{i=1}^n \omega_i (\boldsymbol{\beta}^\top \mathbf{h}(\mathbf{x}_i) - y_i)^2 + \rho\|\boldsymbol{\beta}\|_1 + \lambda\left[\sum_{t=1}^T e_t|\beta_t| + \sum_{\alpha=1}^d c_\alpha \sqrt{\sum_{t=1}^T (F_{\alpha t}\beta_t)^2}\right]. \tag{7}$$

[2] We assume function $\delta(\cdot)$ operates element-wise on vectors and matrices.

[3] Recently, it has been shown that for web-search ranking tasks the squared-loss is surprisingly competitive and improvements from ranking-specific loss functions are typically neglible for non-linear ranking functions such as boosted regression trees (Chapelle and Chang, 2011).

[4] With a slight abuse of notation we will write $\mathcal{L}(\boldsymbol{\beta}), \ell(\boldsymbol{\beta}), r(\boldsymbol{\beta}), c(\boldsymbol{\beta})$ for the functions in (2).

$$\underbrace{\frac{1}{2}\sum_{i=1}^{n}\omega_i\sum_{k=1}^{K}q_i^k\left(y_i-\mathbf{h}(\mathbf{x}_i)^\top\boldsymbol{\beta}^k\right)^2}_{\text{loss}}+\underbrace{\sum_{k=1}^{K}\rho^k\sum_{t=1}^{T}|\beta_t^k|}_{\text{regularization}}+\lambda\left(\underbrace{\sum_{t=1}^{T}e_t\sqrt{\sum_{k=1}^{K}(\beta_t^k d_k)^2}}_{\text{tree-cost}}+\underbrace{\sum_{\alpha=1}^{d}c_\alpha\sqrt{\sum_{k=1}^{K}\sum_{t=1}^{T}(F_{\alpha t}\beta_t^k d_k)^2}}_{\text{feature-cost}}\right) \quad (8)$$

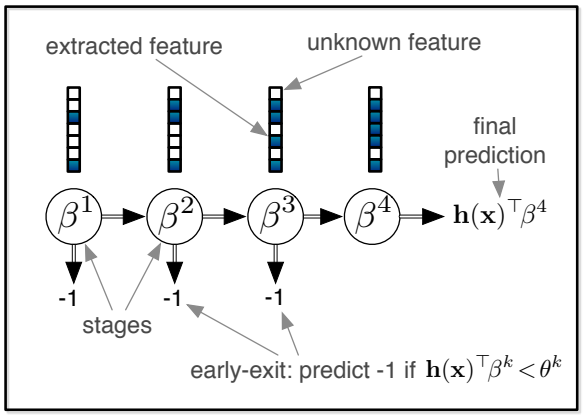

Figure 1: Schematic layout of a classifier cascade with four stages.

3.2 Cascaded Optimization

The previous section has shown how to *re-weight* the trees in order to obtain a solution that balances both accuracy and cost-efficiency. In this section we will go further and *re-order* the trees to allow "easy" inputs to be classified on primarily cheap features and with fewer trees than "difficult" inputs. In our setup, we utilize our assumption that the data set is highly class-skewed. We follow the intuition of Viola and Jones (Viola and Jones, 2002) and stack multiple re-weighted classifiers into an ordered cascade. See Figure 1 for a schematic illustration. Each classifier can reject an input as negative or pass it on to the next classifier. In data sets with only very few positive examples (*e.g.* web-search ranking) such a cascade structure can reduce the average computation time tremendously. Almost all inputs are rejected after only a few cascade steps.

Let us denote a K-stage cascade as $\mathcal{C} = \{(\boldsymbol{\beta}^1,\theta^1),(\boldsymbol{\beta}^2,\theta^2),\cdots,(\boldsymbol{\beta}^K,-)\}$. Each stage has its own weight vector $\boldsymbol{\beta}^k$, which defines a classifier $f^k(\mathbf{x}) = \mathbf{h}(\mathbf{x})^\top\boldsymbol{\beta}^k$. An input is rejected (i.e. classified as negative) at stage k if $\mathbf{h}(\mathbf{x})^\top\boldsymbol{\beta}^k < \theta^k$. The test-time prediction is -1 in case an input is rejected early and otherwise $\mathbf{h}(\mathbf{x})^\top\boldsymbol{\beta}^K$.

Soft assignments. To simulate the early exit of an input $\mathbf{x}$ from the cascade, we define a "soft" indicator function $I_{\boldsymbol{\beta},\theta}(\mathbf{x}) = \sigma_\gamma(\mathbf{h}(\mathbf{x})^\top\boldsymbol{\beta}-\theta)$, where $\sigma_\gamma(\cdot)$ denotes the sigmoid function $\sigma_\gamma(x) = \frac{1}{1+e^{-\gamma x}}$ of steepness $\gamma > 0$. For $\gamma \gg 0$, the function $I_{\boldsymbol{\beta}^k,\theta^k}(\mathbf{x}) \in [0,1]$ approximates the non-continuous $0/1$ step-function indicating whether or not an input $\mathbf{x}$ proceeds beyond stage k (for this writeup we set $\gamma = 50$).

As $I_{\boldsymbol{\beta}^k,\theta^k}(\mathbf{x}_i) \in [0,1]$, we can interpret it as the "probability" that an input $\mathbf{x}_i$ passes stage k, and $p_i^k = \prod_{j=1}^{k-1} I_{\boldsymbol{\beta}^j,\theta^j}(\mathbf{x}_i)$ as the probability that $\mathbf{x}_i$ passes all the stages $1,\ldots,k-1$ prior to k. Further, we let $d_k = \frac{1}{n}\sum_{i=1}^{n} p_i^k$ denote the expected fraction of inputs still in stage k. We can further express the probability that stage k is the *exit-stage* for an input $\mathbf{x}$ as $q_i^k = p_i^k(1-I_{\boldsymbol{\beta}^k,\theta^k})$. (For the last stage, K, we define $q_i^K = p_i^K$, as it is by definition the exit-stage for *every* input that enters it.)

Cascade. In the following, we adapt eq. (7) to this cascade setting. For the sake of clarity, we state the resulting optimization problem in eq. (8) and explain each of the four terms individually.

Loss. The first term in eq. (8) is a direct adaptation of the corresponding term in eq. (7). For every input, the final prediction is computed by its exit-stage. The loss therefore computes the *expected* squared error according to the exit probabilities $q_i^1,\ldots,q_i^K$.

Regularization. Similar to the single stage case, we employ ℓ_1-regularization to avoid over fitting. As the stages differ in number of inputs, we allow a different constant ρ^k per stage. Section 3.4 contains a detailed description on how we set hyper-parameters.

Tree-cost. The third term, $\sum_{t=1}^{T} e_t|\beta_t|$, in eq. (7) addresses the evaluation cost per tree. Naïvely, we could adapt this term as $\sum_{k=1}^{K} d_k \sum_{t=1}^{T} e_t|\beta_t^k|$, where we sum over all trees across all stages – weighted by the fraction of inputs d_k still present in each stage. In reality, it is reasonable to assume that no tree is actually computed twice for the same input[5]. We therefore adapt the same "pricing"-scheme as for features and consider a tree free after its first evaluation. Following a similar reasoning as in section 3.1, we change the ℓ_1-norm to the mixed-norm in order to encourage tree-sparsity *across stages*.

Feature-cost. The transformation of the feature-cost term is analogous to the tree-cost. We introduce two modifications from eq. (7): 1. The mixed-norm is computed *across*

[5]The classifiers in all stages are additive and during test-time we can maintain an accumulator for each stage from the start. If a tree is re-used at a later stage, the appropriately weighted result is added to those stage specific accumulators after its first evaluation. Consequently, each tree is at most evaluated once.

stages. 2. The feature cost per stage is weighted by the expected number of inputs at that stage, d_k.

3.3 Optimization

The optimization is carried out in cycles. In every cycle, K optimization problems are solved, each aiming to learn the classifier weights $\boldsymbol{\beta}^k$ and the early-exit threshold θ^k of a particular stage k to reduce the joint loss of the overall cascade in eq. (8). A detailed pseudo-code implementation is presented in Algorithm 1.

3.3.1 Cyclic Optimization

In each iteration, we minimize the loss with respect to a single stage $(\boldsymbol{\beta}^k, \theta^k)$ at a time – while keeping all other optimization variables fixed.

Loss. With all the weight vectors $\boldsymbol{\beta}^j$ and the early exit thresholds θ^j of the stages $j \neq k$ fixed, for all $j < k$, the variables q_i^j (for $i = 1 \ldots n$) become constants. For subsequent stages $j > k$, q_i^j reduces to $q_i^j = I_{\boldsymbol{\beta}^k,\theta^k}(\mathbf{x}_i)\tilde{q}_i^j$, where $\tilde{q}_i^j$ is a constant. Minimizing the *loss* term in eq. (8) becomes equivalent to minimizing

$$\ell(\boldsymbol{\beta}^k, \theta^k) = const + \frac{1}{2}\sum_{i=1}^{n} \omega_i q_i^k (y_i - \mathbf{h}(\mathbf{x}_i)^\top \boldsymbol{\beta}^k)^2$$
$$+\frac{1}{2}\sum_{i=1}^{n} \underbrace{\left(\omega_i \sum_{j=k+1}^{K} \tilde{q}_i^j \left(y_i - \mathbf{h}(\mathbf{x}_i)^\top \beta^j\right)^2\right)}_{constant} I_{\boldsymbol{\beta}^k,\theta^k}(\mathbf{x}_i) \quad (9)$$

The first term is the regression loss of stages $1 \ldots k-1$, which is a constant. The second term computes the regression loss at stage k and contains $\boldsymbol{\beta}^k$ inside the square and inside the term q_i^k (latter is also constant for $k = K$). The third term accumulates the loss of the stages after stage k, and it depends on $\boldsymbol{\beta}^k$ and θ^k only through $I_{\boldsymbol{\beta}^k,\theta^k}$. As we will point out later in more detail, the third term is the only non-convex part of the loss and vanishes if $k = K$.

Regularization and cost. Before we describe how to perform the optimization, we address how to circumvent the non-differentiability of the $\sqrt{\cdot}$ and $|\cdot|$ operators. The following lemma helps us to rephrase these terms as quadratic forms, which simplifies the optimization greatly. (Please note that $|x| = \sqrt{x^2}$, and therefore we only address the square-root term.)

Lemma 1 *The following holds for any $g(x) > 0$:*

$$\sqrt{g(x)} = \min_{y>0} \frac{1}{2}\left[\frac{g(x)}{z} + z\right]. \quad (10)$$

This lemma is straightforward to prove by noting that the minimizer is $z = \sqrt{g(x)}$. We apply Lemma 1 to (locally) minimize terms of the form $\sqrt{g(\beta_t^k)}$ in the regularization term and cost terms in (8). For this purpose, we introduce auxiliary variables σ_t and κ_t for $1 \leq t \leq T$, and η_α for $1 \leq \alpha \leq d$ which are all minimized, and perform the following substitutions in (8):

$$|\beta_t^k| \rightarrow \frac{1}{2}\left(\frac{(\beta_t^k)^2}{\sigma_t} + \sigma_t\right)$$
$$\sqrt{\sum_{j=1}^{K}(\beta_t^j d_j)^2} \rightarrow \frac{1}{2}\left(\frac{\sum_{j=1}^{K}(\beta_t^j d_j)^2}{\kappa_t} + \kappa_t\right) \quad (11)$$
$$\sqrt{\sum_{j=1}^{K}\sum_{t=1}^{T}(F_{\alpha t}\beta_t^j d_j)^2} \rightarrow \frac{1}{2}\left(\frac{\sum_{j=1}^{K}\sum_{t=1}^{T}(F_{\alpha t}\beta_t^j d_j)^2}{\eta_\alpha} + \eta_\alpha\right)$$

Optimization. Given values for $\boldsymbol{\beta}^k$ and θ^k, the optimal solution σ_t^*, κ_t^*, and η_α^* is available in closed form and consists exactly of the left hand side of the corresponding equation in (11), *e.g.*, $\sigma_t^* = |\beta_t^k|$. Given σ_t^*, κ_t^*, and η_α^*, the resulting loss is continuous and differentiable with respect to $\boldsymbol{\beta}^k$ and θ^k and can be minimized with any off-the-shelf gradient-based optimization package[6]. This alternating optimization with respect to $\boldsymbol{\beta}^k, \theta^k$ and the auxiliary variables σ_t, κ_t, and η_α converges to a local optimum for $\boldsymbol{\beta}^k$ and θ^k. Algorithm 1 summarizes this procedure in pseudo-code.

Algorithm 1 Cyclic Optimization in pseudo-code.

```
Initialize the weight vectors β^k and the early exit thresholds θ^k for all the stages using Algorithm 2.
repeat
    for k = 1 → K do
        Fix β^j, θ^j for j ≠ k.
        repeat
            Fix β^k, θ^k, compute σ*, κ*, η*.
            Fix σ*, κ*, η*, minimize (8) w.r.t. β^k, θ^k.
        until Cost-sensitive loss (8) no longer improves.
    end for
until Cost-sensitive loss (8) no longer improves.
Return the cascade C = {(β^1, θ^1), ··· , (β^K, −)}.
```

3.3.2 Initialization

As the joint loss is not convex, initialization is important. In this section, we describe how to initialize the weight vectors and exit thresholds for the cascade. In particular, we derive a simple *convex* approximation that we use to obtain a good initialization with very little computational effort. Our initialization is based on a simple insight: For $k = K$, all four terms in eq. (8) become convex. We therefore initialize the vectors $\boldsymbol{\beta}^k$ in increasing order of k, each time pretending that the entire cascade consists of only k stages, *i.e.*, $K = k$.

[6]We use `http://tinyurl.com/minimize-m`.

Quadratic forms. For the optimization of a single stage, the *loss*-term in (8) is rewritten as (9). For $k = K$, (9) reduces to a simple quadratic form with respect to $\boldsymbol{\beta}^K$, as its third and only non-quadratic term (the loss of following stages) vanishes. With the help of the following extension to Lemma 1, we can show that all the remaining terms in (8) can also be solved by minimizing quadratic forms:

Lemma 2 *If $g(x) = a + bx^2$ with $a \geq 0$ and $b > 0$, then $\frac{1}{2}\left[\frac{g(x)}{z} + z\right]$ (in Lemma 1) is jointly convex in x and z. Proof, (Boyd and Vandenberghe, 2004, p.72).*

We can show that for $k = K$ Lemma 2 applies to all three cases in (11). It trivially applies to the first one $|\beta_t^K| = \sqrt{(\beta_t^K)^2}$ ($b = 1, a = 0$). In the other two cases, we can show that no d_j is a function of β_t^K. This follows from the definition of d_j and the fact that $j \leq K$. Hence, all expression inside the $\sqrt{\cdot}$ in (11) become quadratic with respect to β_t^k. As all constants are non-negative, and all terms only appear inside squares, Lemma 2 applies. As an example,

$$\sqrt{\sum_{j=1}^{K}(\beta_t^j d_j)^2} = \sqrt{\underbrace{\sum_{j=1}^{K-1}(\beta_t^j d_j)^2}_{a} + \underbrace{d_K^2}_{b}(\beta_t^K)^2}. \quad (12)$$

We follow the same alternate optimization over the auxiliary variables σ, κ, η and the weight vector $\boldsymbol{\beta}^k$ as before, however there are two crucial differences from before: 1. the alternate optimization is now *jointly* convex (see Lemma 2) and will converge to the *global* minimum for this stage. 2. all terms are in quadratic forms and can be solved in *closed form*[7]. In particular, we can collect all constants into two design matrices

$$\Omega_{ii} = \omega_i q_i^k,\ \Lambda_{tt} = \frac{\rho}{\sigma_t} + \lambda {d_k}^2 \left(\frac{e_t}{\kappa_t} + \sum_{\alpha=1}^{d} \frac{c_\alpha F_{\alpha t}}{\eta_\alpha}\right), \quad (13)$$

and solve for $\boldsymbol{\beta}^k$ through

$$\boldsymbol{\beta}^k = (H^\top \Omega H + \Lambda)^{-1} H^\top \Omega \mathbf{y}, \quad (14)$$

with $H_{ti} = h_t(\mathbf{x}_i)$. (For the most common scenario, $n \gg T$, the computation in (14) is dominated by the matrix multiplication $H^\top \Omega H$ — however, because both H and Ω stay constant throughout the optimization of a stage this expression can be pre-computed and re-used.) The initialization is summarized in Algorithm 2 in pseudo-code.

To compensate for the fact that the current stage is not actually the last one, during initialization, we inflate the trade-off parameter λ for early stages and decrease it monotonically until we reach the intended trade-off at the last stage. Here, we follow a simple rule of exponentially decreasing λ, with $\lambda_1 = \lambda\delta^{-K}$ and $\lambda_{k+1} = \lambda_k \delta^{-1}$. As we pretend that stage k is the last stage, and has no exit-threshold during the optimization, we still need to set θ^k after the optimization if $k < K$. Here we choose θ^k such that the expected number of inputs per stage decays by a constant factor $\epsilon > 0$, *i.e.* we set $d_0 = 1$ and θ^k such that $d_k = (1-\epsilon)d_{k-1}$.

Algorithm 2 Initialization in pseudo-code.

```
Input: H = [h(x_i), ∀i ∈ D^k], d_1 = 1.
for k = 1 → K do
    Fix β^j, θ^j for j = 1, ··· , k − 1.
    repeat
        Given β^k, set σ*, κ*, η* to LHS of (11)
        Given σ*, κ*, η*, set β^k with (14)
        Set θ^k so that d_{k+1} = (1−ε)d_k.
    until Loss can no longer be reduced
end for
Return the cascade C_ι = {(β^1, θ^1), ··· , (β^K, θ^k)}.
```

3.4 Hyper-parameters.

The fast initialization procedure provides us with a very efficient way to set hyper-parameters. We initialize with various parameter settings and choose the best-performing setup (without further training) on a validation set.

We use this approach for the ranking task from section 4 to set $\delta = 1.3$ and $\omega^+ = 3.5$ (and by default $\omega^- = 1$). Only the regularization constants of the first and last stages, ρ_1 and ρ_K, are set by cross-validation. For, k such that $1 < k < K$, we set $\rho_k = \rho_1$. The algorithm is surprisingly insensitive to the values of K and ϵ, mostly because if K is too large the optimization can effectively eliminate unnecessary stages by returning all-zero weight vectors $\boldsymbol{\beta}^k$ (which induce zero cost) and negative thresholds θ^k so that all inputs are passed on. We set $K = 10$ and $\epsilon = 0.15$, *i.e.* during initialization each stage removes 15% of the remaining inputs.

4 Results

We conduct experiments on the public Yahoo Learning to Rank Challenge[1] data set. In the original data, the label y_i takes values from $\{0, 1, 2, 3, 4\}$. For simplicity, we only distinguish if a document is relevant ($y_i \geq 3$) to the input query or not and binarize the label accordingly to $y_i \in \{+1, -1\}$. As mentioned, the two classes are unbalanced. Out of the 473134 documents, only 45111 are relevant to the corresponding queries. Although this data set is representative for a web-search ranking *training* set, it is not representative for the *test-case*, in which typically many more negative examples are present. In a real life

[7] This assumes that $\ell(\cdot)$ is the squared-loss. If another convex loss is used, each sub-problem of the initialization is still convex but would have to be solved with gradient descent methods.

[1] Available from http://learningtorankchallenge.yahoo.com

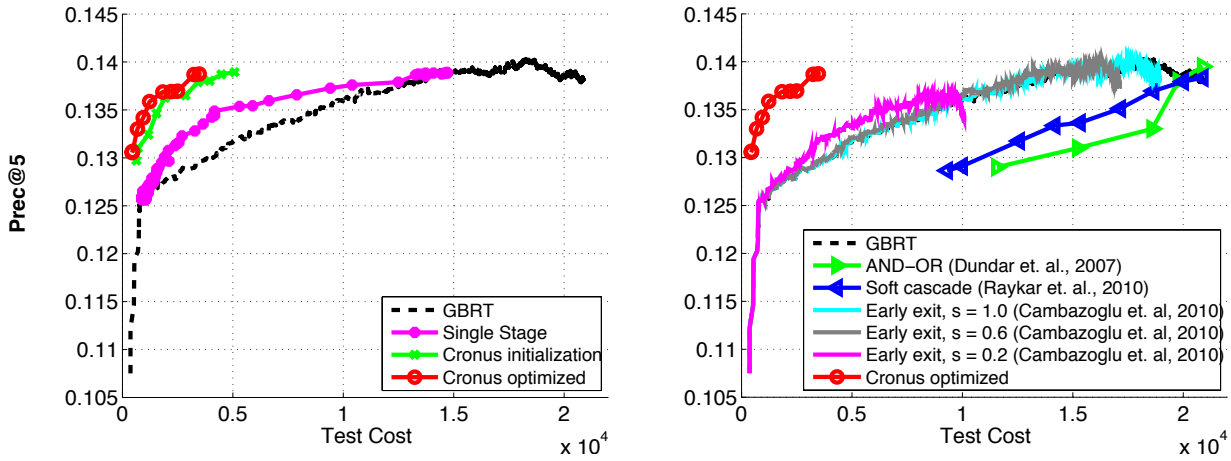

Figure 2: The precision@5 and the test-time cost of various approaches. *Left:* The comparison of the original *GBRT*, single-stage optimization, Cronus after its initialization and optimization (under various settings of λ). There is a clear trend that each step improves the precision-cost curve. *Right:* Comparisons with prior work on test-time optimized cascades. The graph shows the large improvement obtained by Cronus.

test-setting, the distribution of the two classes is even more skewed. Usually for each query, there are only a few documents that are highly relevant, out of hundreds of thousands of candidate documents. The documents in this dataset have in fact been collected to be biased towards relevant documents. Thus, to make our evaluation more realistic, we replicated each negative example 10 times.

For each feature in the dataset, we have a ballpark estimate of its acquisition cost[8]. This cost depends on the feature type and is in the set $\{1, 5, 20, 50, 100, 150, 200\}$. The unit of these costs is approximately the time required for the evaluation of a single tree. The cheapest features (the ones with a cost of 1) are those that can be looked up in a table (such as the statistics of a given document), while the most expensive ones (such as BM25F-SD described in (Broder et al., 2010)) typically involve term proximity scoring.

Many different measures for evaluating the performance of web ranking systems have been proposed, such as NDCG (Järvelin and Kekäläinen, 2002) or ERR (Chapelle et al., 2009). In this work, we use the Precision@5 measurement, i.e., the fraction of the top 5 documents retrieved that are relevant to the query, as it best reflects a classifier's ability to accurately retrieve a small number of relevant instances within a large set of irrelevant documents.

The initial set of trees is constructed using Gradient Boosted Regression Trees (GBRT). Most of the ranking systems implement GBRT or a variation of it.[9] Our implementation of GBRT uses the CART (Breiman et al., 1984; Tyree et al., 2011) algorithm to find the regression tree h_t in each iteration. We end up with a set of $T = 5,000$ trees.

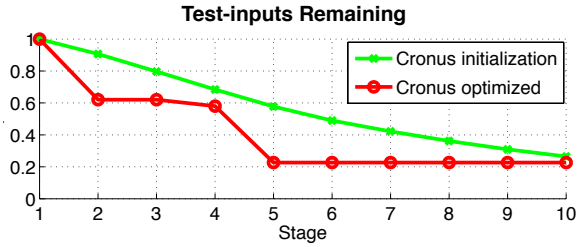

Figure 3: The fraction of test-inputs remaining per stage.

Analysis. We refer to our algorithm as Cronus[10]. Figure 2 (*left*) illustrates the precision/cost performance of the various approaches from this paper (under varying values for λ). The single stage optimization (section 3.1) — which essentially re-weights the trees to minimize cost — significantly improves over the greedy GBRT (dashed black line)[11]. The cascaded Cronus improves substantially over the single-stage approach, even already after its initialization (green curve). Finally, the optimized version of Cronus improves even further and yields the best precision/cost trade-off. The Cronus and single-stage curves were obtained by setting $\lambda = 10^{-4}, 10^{-4}\delta^{-1}, \ldots, 10^{-4}\delta^{-9}$.

Figure 3 shows the fraction of inputs remaining at each stage. The optimized classifier is more aggressive and reduces inputs more rapidly in earlier stages. In fact, a curious artifact of the optimization is that the resulting cascade is reduced to four stages (1,3,4,10). All other weight vectors $\boldsymbol{\beta}^k$ are returned as all-zeros-vectors with a low threshold θ^k that accepts all inputs. Note that these dummy stages have no cost, as they require no trees to be evaluated or fea-

[8]Personal communication with web search experts at Yahoo!

[9]We use the open-source implementation *rt-rank* (Mohan et al., 2011) from `http://tinyurl.com/rtrank`.

[10]According to Greek mythology, in order to gain power, *Cronus* used the help of the *cyclops* — a well-known abbreviation for *cyclic-optimizations*.

[11]The GBRT curve is obtained by plotting the precision and cost as more trees are added.

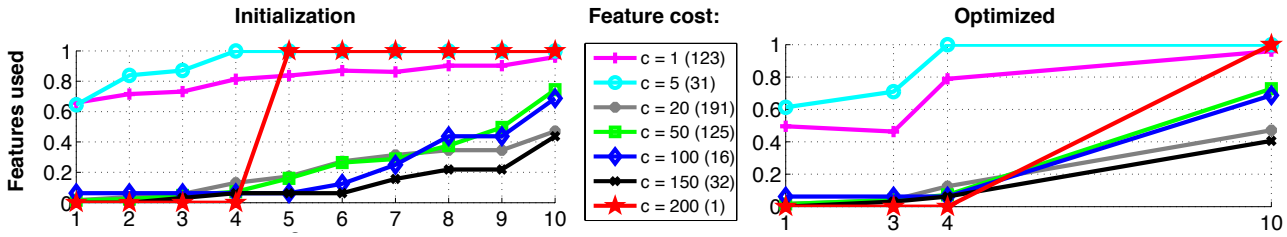

Figure 4: Features (grouped by cost c) used in the various stages of Cronus (the number of features in each cost group is indicated in parentheses in the legend). Most cheap features ($c = 1, 5$) are already extracted in the first stage, whereas expensive features ($c \geq 20$) are only gradually used. (The line for $c = 200$ is a step-function because there exists only a single feature at that cost.)

tures to be extracted, and can be regarded as a way to automatically learn the effective cascade length. Figure 4 depicts what fraction of features of a given cost are extracted at each stage. The number of features for a given cost is indicated in parentheses inside the legend. There is a clear trend that cheap features, $c_\alpha \leq 5$, are extracted early-on while more expensive features $c \geq 20$ are extracted near the end of the cascade if at all. However, it is important to notice that a few expensive features ($c = 100, 150$) are extracted in the very first stage. This highlights one of the great advantages of Cronus over prior work. If a feature is expensive, but very userful, it *can* choose to extract it very early on. This is in strong contrast to (Dundar and Bi, 2007; Raykar et al., 2010), where the features are pre-assigned to stages prior to learning.

Comparison. Figure 2 (*right*) compares Cronus with several state-of-the-art approaches to test-time sensitive learning. *Early exit*, proposed by (Cambazoglu et al., 2010), is identical to GBRT, but the average test-time is reduced by short-circuiting the scoring process of unpromising documents. The authors suggest various methods for early exiting, all of which we implemented. Here we showcase the best-performing method, "Early Exits Using Proximity Threshold", where we introduce an early exit every 10 trees (500 in total) and at the i^{th} early-exit we remove all test-inputs that have a score of at least $\frac{500-i}{499}s$ lower than the fifth best input. Overall, the benefits from early-exiting are very limited in our scenario as the cost is dominated by feature extraction and not tree computation. The approach is limited because it cannot *re-order* the trees such that expensive features are extracted at later stages in the cascade.

Two algorithms, *Soft-Cascade* (Raykar et al., 2010) and *AND-OR* (Dundar and Bi, 2007) provide alternative formulations to globally optimize cascaded classifiers. Both employ an AND-OR learning scheme with a loss function that treats negative instances and positive instances separately. The loss for *positive* instances is the maximum loss across all stages, which corresponds to the AND operation (*i.e.*, *all* stages are encouraged to be correct). For *negative* instance, the loss is a noisy-or (*i.e.,* at least *one* stage must be correct). Feature-cost is accounted for by restricting earlier stages to use only trees with cheaper features. We use five stages in total, allowing features of costs $\leq 5, \leq 20, \leq 50, \leq 150, \leq 200$. The curves were obtained by varying the loss/cost trade-off constant. In addition, the *Soft-Cascade* also incorporates probabilistic thresholding (similar to Cronus). As can be observed in Figure 2, both methods perform rather poorly on the Yahoo! ranking data set. The reason for this is two-fold: 1. several *expensive* features are necessary to obtain high precision test-scores. Because both algorithms assign trees to stages *prior* to learning, trees that use these features are only extracted in late cascade stages, which makes it impossible to obtain high precision at a low cost. 2. both the AND- and OR-loss are order independent and return a low loss even if only late stages obtain high accuracy — therefore magnifying this effect even further by encouraging inputs to only exit in the final cascade stage.

All experiments were conducted on a desktop with dual 6-core Intel i7 cpus with 2.66Ghz. *Early Exit* requires no training and is therefore almost instantaneous. *Soft-Cascade* and *AND-OR* both require several hours for a single training of the Yahoo! data set. With parameter setting by cross-validation, the overall training procedure requires several days. Cronus can be initialized in a few minutes and trained in less than one hour (including cross-validation for the setting of the hyper-parameter).

5 Conclusion

Controlling the operational cost of machine learning algorithms is a crucial problem that appears all-through current and potential applications of machine learning. We believe that understanding and controlling this trade-off will become a fundamental part of machine-learning research in the near future. This paper introduces a novel algorithm, Cronus, to build cascades of classifiers to trade-off prediction accuracy and runtime cost. Different from prior work, our learning framework optimizes the order in which features are extracted globally and provides an elegant and efficient method for initialization and parameter tuning.

References

J. Abernethy, O. Chapelle, and C. Castillo. Web spam identification through content and hyperlinks. In *Proceedings of the 4th international workshop on Adversarial information retrieval on the web*, pages 41–44. ACM, 2008.

J. Bennett and S. Lanning. The netflix prize. In *Proceedings of KDD Cup and Workshop*, volume 2007, page 35, 2007.

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004.

L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and regression trees*. Chapman & Hall/CRC, 1984.

A. Broder, E. Gabrilovich, V. Josifovski, G. Mavromatis, D. Metzler, and J. Wang. Exploiting site-level information to improve web search. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management*, 2010.

B.B. Cambazoglu, H. Zaragoza, O. Chapelle, J. Chen, C. Liao, Z. Zheng, and J. Degenhardt. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 411–420. ACM, 2010.

O. Chapelle and Y. Chang. Yahoo! learning to rank challenge overview. In *Journal of Machine Learning Research, Workshop and Conference Proceedings*, volume 14, pages 1–24, 2011.

O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proceeding of the 18th ACM conference on Information and knowledge management*, pages 621–630, 2009.

O. Chapelle, P. Shivaswamy, S. Vadrevu, K.Q. Weinberger, Ya Zhang, and B. Tseng. Multi-task learning for boosting with application to web search ranking. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1189–1198, 2010.

M.M. Dundar and J. Bi. Joint optimization of cascaded classifiers for computer aided detection. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

M. Fleck, D. Forsyth, and C. Bregler. Finding naked people. *Computer Vision—ECCV'96*, pages 593–602, 1996.

J. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

M. Kowalski. Sparse regression using mixed norms. *Applied and Computational Harmonic Analysis*, 27(3): 303–324, 2009.

L. Lefakis and F. Fleuret. Joint Cascade Optimization Using a Product of Boosted Classifiers. *Advances in neural information processing systems*, 2010.

A. Mohan, Z. Chen, and K.Q. Weinberger. Web-search ranking with initialized gradient boosted regression trees. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:77–89, 2011.

V.C. Raykar, B. Krishnapuram, and S. Yu. Designing efficient cascaded classifiers: tradeoff between accuracy and cost. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 853–860, 2010.

M.J. Saberian and N. Vasconcelos. Boosting classifier cascades. In *Neural Information Processing Systems (NIPS)*, 2010.

R.E. Schapire. A brief introduction to boosting. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1401–1406. Lawrence Erlbaum Associates ltd, 1999.

S. Tyree, K.Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396. ACM, 2011.

P. Viola and M. Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2):137–154, 2002.

L. Wang, J. Lin, and D. Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th Annual International ACM SIGIR conference on Research and development in information retrieval*, 2011.

K.Q. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. In *Advances in Neural Information Processing Systems*. Cambridge, MA, 2008.