Journal of Machine Learning Research 15 (2014) 2113-2144 Submitted 4/13; Revised 12/13; Published 6/14

Classifier Cascades and Trees for Minimizing Feature Evaluation Cost

Zhixiang (Eddie) Xu XUZX@CSE.WUSTL.EDU
Matt J. Kusner MKUSNER@WUSTL.EDU
Kilian Q. Weinberger KILIAN@WUSTL.EDU
Department of Computer Science
Washington University
1 Brookings Drive
St. Louis, MO 63130, USA

Minmin Chen M.CHEN@CRITEO.COM
Olivier Chapelle OLIVIER@CHAPELLE.CC
Criteo
411 High Street
Palo Alto, CA 94301, USA

Editor: Balazs Kegl

Abstract

Machine learning algorithms have successfully entered industry through many real-world applications (*e.g.* , search engines and product recommendations). In these applications, the test-time CPU cost must be budgeted and accounted for. In this paper, we examine two main components of the test-time CPU cost, *classifier evaluation cost* and *feature extraction cost*, and show how to balance these costs with the classifier accuracy. Since the computation required for feature extraction dominates the test-time cost of a classifier in these settings, we develop two algorithms to efficiently balance the performance with the test-time cost. Our first contribution describes how to construct and optimize a tree of classifiers, through which test inputs traverse along individual paths. Each path extracts different features and is optimized for a specific sub-partition of the input space. Our second contribution is a natural reduction of the tree of classifiers into a cascade. The cascade is particularly useful for class-imbalanced data sets as the majority of instances can be early-exited out of the cascade when the algorithm is sufficiently confident in its prediction. Because both approaches only compute features for inputs that benefit from them the most, we find our trained classifiers lead to high accuracies at a small fraction of the computational cost.

Keywords: budgeted learning, resource efficient machine learning, feature cost sensitive learning, web-search ranking, tree of classifiers

1. Introduction

In real-world machine learning applications, such as email-spam (Weinberger et al., 2009), adult content filtering (Fleck et al., 1996), and web-search engines (Zheng et al., 2008; Chapelle et al., 2011), managing the CPU cost at test-time becomes increasingly important. In applications of such large scale, computation must be budgeted and accounted for.

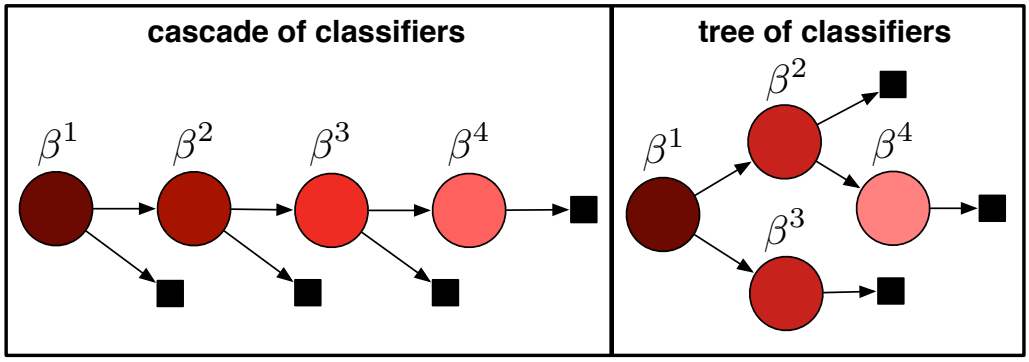

Figure 1: An illustration of two different techniques for learning under a test-time budget. Circular nodes represent classifiers (with parameters $\boldsymbol{\beta}$) and black squares predictions. The color of a classifier node indicates the number of inputs passing through it (darker means more). *Left:* CSCC, a classifier cascade that optimizes the average cost by rejecting easier inputs early. *Right:* CSTC, a tree that trains expert leaf classifiers specialized on subsets of the input space.

Two main components contribute to the *test-time cost*. The time required to evaluate a classifier and the time to extract features used by that classifier. Since the features are often heterogeneous, extraction time for different features is highly variable. Imagine introducing a new feature to a product recommendation system that requires 1 second to extract per recommendation. If a web-service provides 100 million recommendations a day (which is not uncommon), it would require 1200 extra CPU days to extract just this feature. While this additional feature may increase the accuracy of the recommendation system, the cost of computing it for *every recommendation* is prohibitive. This introduces the problem of balancing the test-time cost and the classifier accuracy. Addressing this trade-off in a principled manner is crucial for the applicability of machine learning.

A key observation for minimizing test-time cost is that not all inputs require the same amount of computation to obtain a confident prediction. One celebrated example is face detection in images, where the majority of all image regions do not contain faces and can often be easily rejected based on the response of a few simple Haar features (Viola and Jones, 2004). A variety of algorithms utilize this insight by constructing a cascade of classifiers (Viola and Jones, 2004; Lefakis and Fleuret, 2010; Saberian and Vasconcelos, 2010; Pujara et al., 2011; Chen et al., 2012; Trapeznikov et al., 2013b). Each stage in the cascade can reject an input or pass it on to a subsequent stage. These algorithms significantly reduce the test-time complexity, particularly when the data is *class-imbalanced*, and *few features* are needed to classify instances into a certain class, as in face detection.

Another observation is that it is not only possible that many inputs can be classified correctly using a small subset of all features, but also, such subsets are likely to *vary across inputs*. Particularly for the case in which data is *not* class-imbalanced it may be possible to further lower the test-time cost by extracting fewer, more specialized features per input than the features that would be extracted using a cascade of classifiers. In this paper, we provide a detailed analysis of a new algorithm, *Cost-Sensitive Tree of Classifiers (CSTC)* (Xu et al.,

2013a) that is derived based on this observation. CSTC minimizes an approximation of the exact expected test-time cost required to predict an instance. An illustration of a CSTC tree is shown in the right plot of Figure 1. Because the input space is partitioned by the tree, different features are only extracted where they are most beneficial, and therefore, the average test-time cost is reduced. Unlike prior approaches, which reduce the total cost for every input (Efron et al., 2004) or which combine feature cost with mutual information to select features (Dredze et al., 2007), a CSTC tree incorporates *input-dependent feature selection* into training and dynamically allocates higher feature budgets for infrequently traveled tree-paths.

CSTC incorporates two novelties: 1. it relaxes the expected per-instance test-time cost into a well-behaved optimization; and 2. it is a generalization of cascades to trees. Full trees, however, are not always necessary. In data scenarios with highly skewed class imbalance, cascades might be a better model by rejecting many instances using a small number of features. We therefore apply the same test-time cost derivation to a stage-wise classifier for cascades. The resulting algorithm, *Cost-Sensitive Cascade of Classifiers (CSCC)*, is shown in the left plot of Figure 1. This algorithm supersedes an approach previously proposed, *Cronus* (Chen et al., 2012), which is not derived through a formal relaxation of the test-time cost, but performs a clever weighting scheme. We compare and contrast Cronus with CSCC.

Two earlier short papers already introduce CSTC (Xu et al., 2013a) and Cronus (Chen et al., 2012) algorithms, however the present manuscript provides significantly more detailed analysis, experimental results and insightful discussion—and it introduces CSCC, which combines insights from all prior work. The paper is organized as follows. Section 2 introduces and defines the test-time cost learning setting. Section 3 presents the tree of classifiers approach, CSTC. In Section 4 we lay out CSCC and relate it to prior work, Cronus, (Chen et al., 2012). Section 5 introduces non-linear extensions to CSTC and CSCC. Section 6 presents the experimental results on several data sets and discusses the performance differences. Section 7 reviews the prior and related contributions that inspires our work. We conclude in Section 8 by summarizing our contributions and proposing a few future directions.

2. Test-Time Cost

There are several key aspects towards learning under test-time cost budgets that need to be considered: 1. feature extraction cost is relevant and varies significantly across features; 2. features are extracted *on-demand* rather than prior to evaluation; 3. different features can be extracted for different inputs; 4. the test cost is evaluated in average rather than in absolute (/worst-case) terms (*i.e.* , several cheap classifications can free up budget for an expensive classification). In this section we focus on learning a single cost-sensitive classifier. We will combine these classifiers to form our tree and cascade algorithms in later sections. We first introduce notation and our general setup, and then provide details on how we address these specific aspects.

Let the data consist of inputs $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathcal{R}^d$ with corresponding class labels $\{y_1, \ldots, y_n\} \subseteq \mathcal{Y}$, where $\mathcal{Y} = \mathcal{R}$ in the case of regression ($\mathcal{Y}$ could also be a finite set of categorical labels). We summarize all notation in Table 1.

$\mathbf{x}_i$	Input instance i
y_i	Input label i
$\mathcal{H}$	Set of all weak learner t
H	Linear classifier on input
$\boldsymbol{\beta}$	Parameters of linear classifier H
ℓ	Non-negative loss function over input
ρ	Coefficient for regularization
λ	Accuracy/cost trade-off parameter
c_α	Feature extraction cost of feature α
v^k	Classifier node k
θ^k	Splitting threshold of node v^k
p_i^k	Traversal probability to node v^k of input $\mathbf{x}_i$
π^l	Set of classifier node along the path from root to v^l

Table 1: Notation used throughout this manuscript.

2.1 Cost-Sensitive Loss Minimization

We learn a classifier $H : \mathcal{R}^d \to \mathcal{Y}$, parameterized by $\boldsymbol{\beta}$, to minimize a continuous, non-negative loss function ℓ over $\mathcal{D}$,

$$\frac{1}{n} \min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \ell(H(\mathbf{x}_i; \boldsymbol{\beta}), y_i).$$

We assume that H is a linear classifier, $H(\mathbf{x}; \boldsymbol{\beta}) = \boldsymbol{\beta}^\top \mathbf{x}$. To avoid overfitting, we deploy a standard l_1 regularization term, $|\boldsymbol{\beta}|$ to control model complexity. This regularization term has the known side-effect to keep $\boldsymbol{\beta}$ sparse (Tibshirani, 1996), which requires us to only evaluate a subset of all features. In addition, to balance the test-time cost incurred by the classifier, we also incorporate the cost term $c(\boldsymbol{\beta})$ described in the following section. The combined test-time cost-sensitive optimization becomes

$$\min_{\boldsymbol{\beta}} \underbrace{\sum_i \ell(\mathbf{x}_i^\top \boldsymbol{\beta}, y_i) + \rho|\boldsymbol{\beta}|}_{\text{regularized loss}} \; + \; \lambda \underbrace{c(\boldsymbol{\beta})}_{\text{test-cost}}, \tag{1}$$

where λ is the accuracy/cost trade-off parameter, and ρ controls the strength of the regularization.

2.2 Test-Time Cost

The test-time cost of H is regulated by the features extracted for that classifier. Different from traditional settings, where all features are computed prior to the application of H, we assume that features are computed *on demand* the first time they are used.

We denote the extraction cost for feature α as c_α. The cost $c_\alpha \geq 0$ is suffered at most once, only for the initial extraction, as feature values can be cached for future use. For a classifier H, parameterized by $\boldsymbol{\beta}$, we can record the features used:

$$\|\beta_\alpha\|_0 = \begin{cases} 1 & \text{if feature } \alpha \text{ is used in } H \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Here, $\|\cdot\|_0$ denotes the l_0 norm with $\|a\|_0 = 1$ if $a \neq 0$ and $\|a\|_0 = 0$ otherwise. With this notation, we can formulate the total test-time cost required to evaluate a test input $\mathbf{x}$ with classifier H (and parameters $\boldsymbol{\beta}$) as

$$c(\boldsymbol{\beta}) = \sum_{\alpha=1}^{d} c_\alpha \|\beta_\alpha\|_0. \tag{3}$$

The equation (3) can be in any units of cost. For example in medical applications, the feature extraction cost may be in units of "patient agony" or in "examination cost". The current formulation (1) with cost term (3) still extracts the same features for all inputs and is NP-hard to optimize (Korte and Vygen, 2012, Chapter 15). We will address these issues in the following sections.

3. Cost-Sensitive Tree of Classifiers

We introduce an algorithm that is inspired by the observation that many inputs could be classified correctly based on only a small subset of all features, and this subset may vary across inputs. Our algorithm employs a tree structure to extract particular features for particular inputs, and we refer to it as the *Cost-Sensitive Tree of Classifiers* (*CSTC*). We begin by introducing foundational concepts regarding the CSTC tree and derive a global cost term that extends (3) to trees of classifiers and then we relax the resulting loss function into a well-behaved optimization problem.

3.1 CSTC Nodes

The fundamental building block of the CSTC tree is a CSTC node—a linear classifier as described in Section 2.1. Our classifier design is based on the assumption that instances with similar labels tend to have similar features. Thus, we design our tree algorithm to partition the input space based on classifier predictions. Intermediate classifiers determine the path of instances through the tree and leaf classifiers become experts for a small subset of the input space.

Correspondingly, there are two different nodes in a CSTC tree (depicted in Figure 2): *classifier nodes* (white circles) and *terminal elements* (black squares). Each *classifier node* v^k is associated with a weight vector $\boldsymbol{\beta}^k$ and a threshold θ^k. These classifier nodes branch inputs by their threshold θ^k, sending inputs to their upper child if $\mathbf{x}_i^\top \boldsymbol{\beta}^k > \theta^k$, and to their lower child otherwise. *Terminal elements* are "dummy" structures and are *not* real classifiers. They return the predictions of their direct parent classifier nodes—essentially functioning as a placeholder for an exit out of the tree. The tree structure may be a full balanced binary tree of some depth (e.g., Figure 2), or can be pruned based on a validation set. For simplicity, we assume at this point that nodes with terminal element children must be leaf nodes (as depicted in the figure)—an assumption that we will relax later on.

During test-time, inputs traverse through the tree, starting from the root node v^0. The root node produces predictions $\mathbf{x}_i^\top \boldsymbol{\beta}^0$ and sends the input $\mathbf{x}_i$ along one of two different paths, depending on whether $\mathbf{x}_i^\top \boldsymbol{\beta}^0 > \theta^0$. By repeatedly branching the test inputs, classifier nodes sitting deeper in the tree only handle a small subset of all inputs and become specialized towards that subset of the input space.

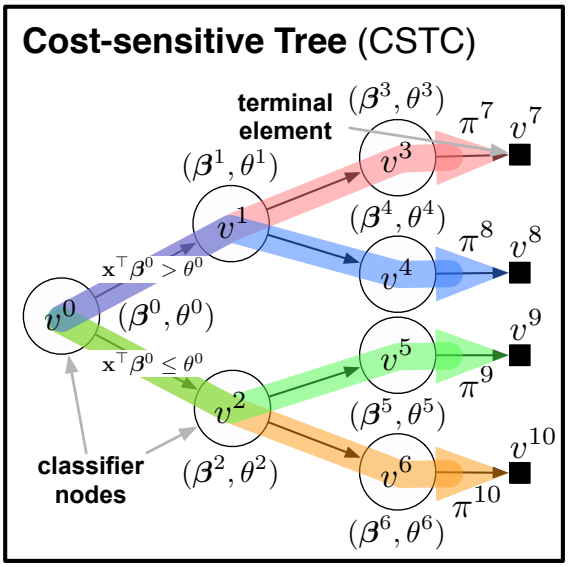

Figure 2: A schematic layout of a CSTC tree. Each node v^k is associated with a weight vector $\boldsymbol{\beta}^k$ for prediction and a threshold θ^k to send instances to different parts of the tree. We solve for $\boldsymbol{\beta}^k$ and θ^k that best balance the accuracy/cost trade-off for the whole tree. All paths of a CSTC tree are shown in color.

3.2 CSTC Loss

In this section, we discuss the loss and test-time cost of a CSTC tree. We then derive a single global loss function over all nodes in the CSTC tree.

3.2.1 Soft Tree Traversal

As we described before, inputs are partitioned at each node during test-time, and we use a hard threshold to achieve this partitioning. However, modeling a CSTC tree with hard thresholds leads to a combinatorial optimization problem that is NP-hard (Korte and Vygen, 2012, Chapter 15). As a remedy, during training, we *softly* partition the inputs and assign *traversal probabilities* $p(v^k|\mathbf{x}_i)$ to denote the likelihood of input $\mathbf{x}_i$ traversing through node v^k. Every input $\mathbf{x}_i$ traverses through the root, so we define $p(v^0|\mathbf{x}_i) = 1$ for all i. We define a "sigmoidal" soft belief that an input $\mathbf{x}_i$ will transition from classifier node v^k with threshold θ^k to its *upper* child v^u as

$$p(v^u|\mathbf{x}_i, v^k) = \frac{1}{1 + \exp(-(\mathbf{x}_i^\top \boldsymbol{\beta}^k - \theta^k))}. \tag{4}$$

Let v^k be a node with upper child v^u and lower child v^l. We can express the probabilities of reaching nodes v^u and v^l recursively as $p(v^u|\mathbf{x}_i) = p(v^u|\mathbf{x}_i, v^k)p(v^k|\mathbf{x}_i)$ and $p(v^l|\mathbf{x}_i) = \big[1 - p(v^u|\mathbf{x}_i, v^k)\big]p(v^k|\mathbf{x}_i)$ respectively. Note that it follows immediately, that if $\mathcal{V}^d$ contains

all nodes at tree-depth d, we have

$$\sum_{v \in \mathcal{V}^d} p(v|\mathbf{x}) = 1. \tag{5}$$

In the following paragraphs we incorporate this probabilistic framework into the loss and cost terms of (1) to obtain the corresponding *expected* tree loss and tree cost.

3.2.2 Expected Tree Loss

To obtain the *expected tree loss*, we sum over all nodes $\mathcal{V}$ in a CSTC tree and all inputs and weight the loss $\ell(\cdot)$ of input $\mathbf{x}_i$ at each node v^k by the probability that the input reaches v^k, $p_i^k = p(v^k|\mathbf{x}_i)$,

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{v^k \in \mathcal{V}} p_i^k \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i). \tag{6}$$

This has two effects: 1. the local loss for each node focuses more on likely inputs; 2. the global objective attributes more weight to classifiers that serve many inputs. Technically, the prediction of the CSTC tree is made entirely by the terminal nodes (*i.e.* , the leaves), and an obvious suggestion may be to only minimize their classification losses and leave the interior nodes as "gates" without any predictive abilities. However, such a setup creates local minima that send all inputs to the terminal node with the lowest initial error rate. These local minima are hard to escape from and therefore we found it to be important to minimize the loss for *all* nodes. Effectively, this forces a structure onto the tree that similarly labeled inputs leave through similar leaves and achieves robustness by assigning high loss to such pathological solutions.

3.2.3 Expected Tree Costs

The cost of a test input is the cumulative cost across all classifiers along its path through the CSTC tree. Figure 2 illustrates an example of a CSTC tree with all paths highlighted in color. Every test input must follow along exactly one of the paths from the root to a terminal element. Let $\mathcal{L}$ denote the set of all terminal elements (*e.g.*, in Figure 2 we have $\mathcal{L} = \{v^7, v^8, v^9, v^{10}\}$), and for any $v^l \in \mathcal{L}$ let π^l denote the set of all *classifier nodes* along the unique path from the root v^0 before terminal element v^l (*e.g.*, $\pi^9 = \{v^0, v^2, v^5\}$).

For an input $\mathbf{x}$, exiting through terminal node v^l, a feature α needs to be extracted if and only if at least one classifier along the path π^l uses this feature. We extend the indicator function defined in (2) accordingly:

$$\left\| \sum_{v^j \in \pi^l} |\beta_\alpha^j| \right\|_0 = \begin{cases} 1 & \text{if feature } \alpha \text{ is used along path to terminal node } v^l \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

We can extend the cost term in (3) to capture the traversal cost from root to node v^l as

$$c^l = \sum_\alpha c_\alpha \left\| \sum_{v^j \in \pi^l} |\beta_\alpha^j| \right\|_0. \tag{8}$$

Given an input $\mathbf{x}_i$, the expected cost is then $E[c^l|\mathbf{x}_i] = \sum_{l\in\mathcal{L}} p(v^l|\mathbf{x}_i)c^l$. To approximate the data distribution, we sample uniformly at random from our training set, *i.e.* , we set $p(\mathbf{x}_i) \approx \frac{1}{n}$, and obtain the unconditional expected cost

$$E[cost] = \sum_{i=1}^{n} p(\mathbf{x}_i) \sum_{l\in\mathcal{L}} p(v^l|\mathbf{x}_i)c^l \approx \sum_{l\in\mathcal{L}} c^l \underbrace{\sum_{i=1}^{n} p(v^l|\mathbf{x}_i)\frac{1}{n}}_{:=p^l} = \sum_{l\in\mathcal{L}} c^l p^l. \tag{9}$$

Here, p^l denotes the probability that a randomly picked training input exits the CSTC tree through terminal node v^l. We can combine (8), (9) with (6) and obtain the objective function,

$$\sum_{v^k\in\mathcal{V}} \underbrace{\left(\frac{1}{n}\sum_{i=1}^{n} p_i^k \ell_i^k + \rho|\boldsymbol{\beta}^k|\right)}_{\text{regularized loss}} + \lambda \sum_{v^l\in\mathcal{L}} p^l \underbrace{\left[\sum_{\alpha} c_\alpha \left\| \sum_{v^j\in\pi^l} |\beta_\alpha^j| \right\|_0\right]}_{\text{test-time cost}}, \tag{10}$$

where we use the abbreviations $p_i^k = p(v^k|\mathbf{x}_i)$ and $\ell_i^k = \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$.

3.3 Test-Cost Relaxation

The cost penalties in (10) are exact but difficult to optimize due to the discontinuity and non-differentiability of the l_0 norm. As a solution, throughout this paper we use the mixed-norm relaxation of the l_0 norm over sums,

$$\sum_j \left\| \sum_i |A_{ij}| \right\|_0 \rightarrow \sum_j \sqrt{\sum_i (A_{ij})^2}, \tag{11}$$

described by Kowalski (2009). Note that for a vector, this relaxes the l_0 norm to the l_1 norm, *i.e.* , $\sum_j \|a_j\|_0 \rightarrow \sum_j \sqrt{(a_j)^2} = \sum_j |a_j|$, recovering the commonly used approximation to encourage sparsity. For matrices $\mathbf{A}$, the mixed norm applies the l_1 norm over rows and the l_2 norm over columns, thus encouraging a whole row to be all-zero or non-sparse. In our case this has the natural interpretation to encourage re-use of features that are already extracted along a path. Using the relaxation in (11) on the l_0 norm in (10) gives the final optimization problem:

$$\min_{\boldsymbol{\beta}^0,\theta^0,\ldots,\boldsymbol{\beta}^{|V|},\theta^{|V|}} \sum_{v^k\in\mathcal{V}} \underbrace{\left(\frac{1}{n}\sum_{i=1}^{n} p_i^k \ell_i^k + \rho|\boldsymbol{\beta}^k|\right)}_{\text{regularized loss}} + \lambda \sum_{v^l\in\mathcal{L}} p^l \underbrace{\left[\sum_{\alpha} c_\alpha \sqrt{\sum_{v^j\in\pi^l} (\beta_\alpha^j)^2}\right]}_{\text{test-time cost penalty}} \tag{12}$$

We can illustrate the fact that the mixed-norm encourages re-use of features with a simple example. If two classifiers $v^k \neq v^{k'}$ along a path π^l use *different* features with identical weight, *i.e.* , $\beta_t^k = \epsilon = \beta_s^{k'}$ and $t \neq s$, the test-time cost penalty along π^l is $\sqrt{\epsilon^2} + \sqrt{\epsilon^2} = 2\epsilon$. However, if the two classifiers *re-use* the same feature, *i.e.* , $t = s$, the test-time cost penalty reduces to $\sqrt{\epsilon^2 + \epsilon^2} = \sqrt{2}\epsilon$.

3.4 Optimization

There are many techniques to minimize the objective in (12). We use block coordinate descent, optimizing with respect to the parameters of a single classifier node v^k at a time, keeping all other parameters fixed. We perform a level order tree traversal, optimizing each node in order: $v^1, v^2, \ldots, v^{|V|}$. To minimize (12) (up to a local minimum) with respect to parameters $\boldsymbol{\beta}^k, \theta^k$ we use the lemma below to overcome the non-differentiability of the square-root term (and l_1 norm, which we can rewrite as $|a| = \sqrt{a^2}$) resulting from the l_0-relaxation (11).

Lemma 1. *Given a positive function* $g(x)$, the following holds:

$$\sqrt{g(x)} = \inf_{z>0} \frac{1}{2}\left[\frac{g(x)}{z} + z\right]. \tag{13}$$

It is straight-forward to see that $z = \sqrt{g(x)}$ minimizes the function on the right hand side and satisfies the equality, which leads to the proof of the lemma.

For each square-root or l_1 term we 1) introduce an auxiliary variable (*i.e.*, z above), 2) substitute in (13), and 3) alternate between minimizing the objective in (12) with respect to $\boldsymbol{\beta}^k, \theta^k$ and solving for the auxiliary variables. The former minimization is performed with conjugate gradient descent and the latter can be computed efficiently in closed form. This pattern of block-coordinate descent followed by a closed form minimization is repeated until convergence. Note that the objective is guaranteed to converge to a fixed point because each iteration decreases the objective function, which is bounded below by zero. In the following subsection, we detail the block coordinate descent optimization technique. Lemma 1 is only defined for strictly positive functions $g(x)$. As we are performing function minimization, we can reach cases where $g(x) = 0$ and Lemma 1 is ill defined. Thus, as a practical work-around, we clamp values to zero once they are below a small threshold (10^{-4}).

3.4.1 Optimization Details

For reproducibility, we describe the optimization in more detail. Readers not interested in the exact procedure may skip to Section 3.5. As terminal nodes are only placeholders and do not have their own parameters, we only focus on classifier nodes, which are depicted as round circles in Figure 2.

Leaf Nodes. The optimization of leaf nodes (*e.g.* , v^3, v^4, v^5, v^6 in Fig. 2) is simpler because there are no downstream dependencies. Let v^k be such a classifier node with only a single "dummy" terminal node $v^{k'}$. During optimization of (12), we fix all other parameters $\boldsymbol{\beta}^j, \theta^j$ of other nodes v^j and the respective terms become constants. Therefore, we remove all other paths, and only minimize over the path $\pi^{k'}$ from the root to terminal node $v^{k'}$. Even along the path $\pi^{k'}$ most terms become constant and the only non-constant parameter is $\boldsymbol{\beta}^k$ (the branching parameter θ^k can be set to $-\infty$ because v^k has only one child). We color non-constant terms in the remaining function in blue below,

$$\sum_i p_i^k \ell(\phi(\mathbf{x}_i)^\top \boldsymbol{\beta}^k, y_i) + \rho|\boldsymbol{\beta}^k| + \lambda\, p^{k'}\left[\sum_\alpha c_\alpha \sqrt{(\beta_\alpha^k)^2 + \sum_{v^j \in \pi^{k'} \setminus v^k} (\beta_\alpha^j)^2}\right], \tag{14}$$

where $\mathcal{S}\backslash b$ contain all of the elements in $\mathcal{S}$ except b. After identifying the non-constant terms, we can apply Lemma 1, making (14) differentiable with respect to β_α^k. Let us define auxiliary variables γ_α and η_α for $1 \leq \alpha \leq d$ for the l_1-regularization term and the test-time cost term. Further, let us collect the constants in the test-time cost term $c_{\text{test-time}} = \sum_{v^j \in \pi^{k'} \setminus v^k} (\beta_\alpha^j)^2$. Applying Lemma 1 results in the following substitutions:

$$\sum_\alpha \rho |\beta_\alpha^k| = \sum_\alpha \rho \sqrt{(\beta_\alpha^k)^2} \longrightarrow \sum_\alpha \rho \frac{1}{2} \left(\frac{(\beta_\alpha^k)^2}{\gamma_\alpha} + \gamma_\alpha \right),$$
$$\sum_\alpha c_\alpha \sqrt{(\beta_\alpha^k)^2 + c_{\text{test-time}}} \longrightarrow \sum_\alpha c_\alpha \frac{1}{2} \left(\frac{(\beta_\alpha^k)^2 + c_{\text{test-time}}}{\eta_\alpha} + \eta_\alpha \right). \tag{15}$$

As a result, we obtain a differentiable objective function after making the above substitutions. We can solve $\boldsymbol{\beta}^k$ by alternately minimizing the obtained differentiable function w.r.t. $\boldsymbol{\beta}^k$ with $\gamma_\alpha, \eta_\alpha$ fixed, and minimizing $\gamma_\alpha, \eta_\alpha$ with $\boldsymbol{\beta}^k$ fixed (*i.e.*, minimizing η_α is equivalent to setting $\eta_\alpha = \sqrt{(\beta_\alpha^k)^2 + c_{\text{test-time}}}$). Recall that θ^k does not require optimization as v^k does not further branch inputs.

It is straight-forward to show (Boyd and Vandenberghe, 2004, page 72), that the right hand side of Lemma 1 is jointly convex in x and z, so as long as $g(x)$ is a quadratic function of x. Thus, if $\ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$ is the squared loss, the substituted objective function is jointly convex in $\boldsymbol{\beta}^k$ and in $\gamma_\alpha, \eta_\alpha$ and therefore we can obtain a globally-optimal solution. Moreover, we can solve $\boldsymbol{\beta}^k$ in closed form. Let us define three design matrices

$$\mathbf{X}_{i\alpha} = [\mathbf{x}_i]_\alpha, \quad \boldsymbol{\Omega}_{ii} = p_i^k, \quad \boldsymbol{\Gamma}_{\alpha\alpha} = \frac{\rho}{\gamma_\alpha} + \lambda \left(\frac{p^k c_\alpha}{\eta_\alpha} \right),$$

where $\boldsymbol{\Omega}$ and $\boldsymbol{\Gamma}$ are both diagonal and $[\mathbf{x}_i]_\alpha$ is the α feature of instance $\mathbf{x}_i$. The closed-form solution for $\boldsymbol{\beta}^k$ is as follows,

$$\boldsymbol{\beta}^k = (\mathbf{X}^\top \boldsymbol{\Omega} \mathbf{X} + \boldsymbol{\Gamma})^{-1} \mathbf{X}^\top \boldsymbol{\Omega} \mathbf{y}. \tag{16}$$

Intermediate Nodes. We further generalize this approach to all classifier nodes. As before, we optimize one node at a time, fixing the parameters of all other nodes. However, optimizing the parameters $\boldsymbol{\beta}^k, \theta^k$ of an *internal* node v^k, which has two children affects the parameters of descendant nodes. This affects the optimization of the regularized classifier loss and the test-time cost separately. We state how these terms in the global objective (12) are affected, and then show how to minimize it.

Let $\mathcal{S}$ be the set containing all descendant nodes of v^k. Changes to the parameters $\boldsymbol{\beta}^k, \theta^k$ will affect the traversal probabilities p_i^j for all $v^j \in \mathcal{S}$ and therefore enter the downstream loss functions. We first state the regularized loss part of (12) and once again color non-constant parameters in blue,

$$\frac{1}{n} \sum_i p_i^k \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i) + \frac{1}{n} \sum_{v^j \in \mathcal{S}} \sum_i p_i^j \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^j, y_i) + \rho |\boldsymbol{\beta}^k|. \tag{17}$$

For the cost terms in (12), recall that the cost of each path π^l is weighted by the probability p^l of traversing that path. Changes to $\boldsymbol{\beta}^k, \theta^k$ affect the probability of any path

Algorithm 1 CSTC global optimization

Input: data $\{\mathbf{x}_i, y_i\} \in \mathcal{R}^d \times \mathcal{R}$, initialized CSTC tree
repeat
  for $k = 1$ **to** $N = \#$ CSTC nodes **do**
    repeat
      Solve for γ, η (fix $\boldsymbol{\beta}^k, \theta^k$) using left hand side of (15)
      Solve for $\boldsymbol{\beta}^k, \theta^k$ (fix γ, η) with conjugate gradient descent, or in closed-form
    until objective changes less than ε
  end for
until objective changes less than ϵ

that passes through v^k and its corresponding probability p^l. Let $\mathcal{P}$ be the terminal elements associated with paths passing through v^k. We state the cost function with non-constant parameters in blue,

$$\sum_{v^l \in \mathcal{P}} p^l \underbrace{\left[\sum_{\alpha} c_\alpha \sqrt{\left(\sum_{v^j \in \pi^l \setminus v^k} (\beta_\alpha^j)^2 + (\beta_\alpha^k)^2 \right)} \right]}_{\text{test-time cost}} \tag{18}$$

Adding (17) and (18), with the latter weighted by λ, gives the internal node loss. To make the combined objective function differentiable we apply Lemma 1 to the l_1-regularization, and test-time cost terms and introduce auxiliary variables $\gamma_\alpha, \eta_\alpha$ as in (15). Similar to the leaf node case, we solve $\boldsymbol{\beta}^k, \theta^k$ by alternately minimizing the new objective w.r.t. $\boldsymbol{\beta}^k, \theta^k$ with $\gamma_\alpha, \eta_\alpha$ fixed, and minimizing $\gamma_\alpha, \eta_\alpha$ with fixed $\boldsymbol{\beta}^k, \theta^k$. Unlike leaf nodes, optimizing the objective function w.r.t. $\boldsymbol{\beta}^k, \theta^k$ cannot be expressed in closed form even with squared loss. Therefore, we optimize it with conjugate gradient descent. Algorithm 1 describes how the entire CSTC tree is optimized.

3.4.2 Node Initialization

The minimization of (12) is non-convex and is therefore initialization dependent. However, minimizing (12) with respect to the parameters of leaf classifiers *is convex*. We therefore initialize the tree top-to-bottom, starting at v^0, and optimizing over $\boldsymbol{\beta}^k$ by minimizing (12) while considering all descendant nodes of v^k as "cut-off" (thus pretending node v^k is a leaf). This initialization is also very fast in the case of a quadratic loss, as it can be solved for in closed form.

3.5 Fine-Tuning

The original test-time cost term in (3) sums over the cost of all features that are extracted during test-time. The relaxation in (11) makes the exact l_0 cost differentiable and is still well suited to *select which* features to extract. However, the mixed-norm does also impact the performance of the classifiers, because (different from the l_0 norm) larger weights in $\boldsymbol{\beta}$ incur larger cost penalties. We therefore introduce a post-processing step to correct the classifiers from this unwanted regularization effect. We re-optimize the loss of all *leaf* classifiers (*i.e.*

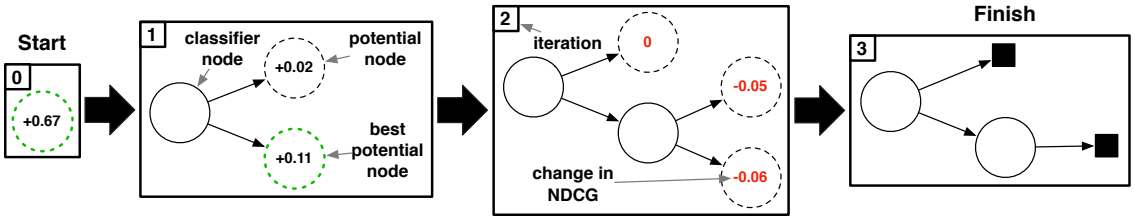

Figure 3: A schematic layout of the greedy tree building algorithm. Each iteration we add the best performing potential node (dashed, above) to the tree. Each potential node is annotated by the improvement in validation-NDCG, obtained with its inclusion (number inside the circle). In this example, after two iterations no more nodes improve the NDCG and the algorithm terminates, converting all remaining potential nodes into terminal elements (black boxes).

, classifiers that make final predictions), while clamping all features with zero-weight to strictly remain zero.

$$\min_{\bar{\boldsymbol{\beta}}^k} \sum_i p_i^k \ell(\mathbf{x}_i^\top \bar{\boldsymbol{\beta}}^k, y_i) + \rho|\bar{\boldsymbol{\beta}}^k|$$
$$\text{subject to: } \bar{\beta}_t^k = 0 \text{ if } \beta_t^k = 0.$$

Here, we do not include the cost-term, because the decision regarding which features to use is already made. The final CSTC tree uses these re-optimized weight vectors $\bar{\boldsymbol{\beta}}^k$ for all leaf classifier nodes v^k.

3.6 Determining the Tree Structure

As the CSTC tree does not need to be balanced, its structure is an implicit parameter of the algorithm. We learn and fix the tree structure prior to the optimization and fine-tuning steps in Sections 3.4 and 3.5. We discuss two approaches to determine the structure of the tree in the absence of prior knowledge, the first prunes a balanced tree bottom-up, the second adds nodes top-down, only when necessary. In practice, both techniques produce similar results and we settled on using the pruning technique for all of our experiments.

3.6.1 Tree Pruning

We build a full (balanced) CSTC tree of depth d and initialize all nodes. To obtain a more compact model and to avoid over-fitting, the CSTC tree can be pruned with the help of a validation set. We compute the validation error of the initialized CSTC tree at each node. Starting with the leaf nodes, we then prune away nodes that, upon removal, do not decrease the validation performance (in the case of ranking data, we even can use validation NDCG (Järvelin and Kekäläinen, 2002) as our pruning criterion). After pruning, the tree structure is fixed and all nodes are optimized with the procedure described in Section 3.4.1.

3.6.2 Greedy Tree Building

In contrast to the bottom-up pruning, we can also use a top-down approach to construct the tree structure. Figure 3 illustrates our greedy heuristic for CSTC tree construction. In each iteration, we add the child node that improves the validation criteria (*e.g.*, NDCG) the most on the validation set.

More formally, we distinguish between CSTC *classifier nodes* and *potential nodes*. Potential nodes (dotted circles in Figure 3) can turn into classifier nodes or terminal elements. Each potential node is initially a trained classifier and annotated with the NDCG value that the CSTC tree would reach on validation with its inclusion. At iteration 0 we learn a single CSTC node by minimizing (12) for the root node v^0, and make it a *potential node*. At iteration $i > 0$ we pick the potential node whose inclusion improves the validation NDCG the most (depicted as the dotted green circle) and add it to the tree. Then we create two new potential nodes as its children, and initialize their classifiers by minimizing (12) with all other weight-vectors and thresholds fixed. The splitting threshold θ^k is set to move 50% of the validation inputs to the upper child (the thresholds will be re-optimized subsequently). This procedure continues until no more potential nodes improve the validation NDCG, and we convert all remaining potential nodes into terminal elements.

4. Cost-Sensitive Cascade of Classifiers

Many real world applications have data distributions with high class imbalance. One example is face detection, where the vast majority of all image patches does not contain faces; another example is web-search ranking, where almost all web-pages are irrelevant to a given query. Often, a few features may suffice to detect that an image does not contain a face or that a web-page is irrelevant. Further, in applications such as web-search ranking, the accuracy of bottom ranked instances is irrelevant as long as they are not retrieved at the top (and therefore are not displayed to the end user).

In these settings, the entire focus of the algorithm should be on the most confident positive samples. Sub-trees that lead to only negative predictions, can be pruned effectively as there is no value in providing fine-grained differentiation between negative samples. This further reduces the average feature cost, as negative inputs traverse through shorter paths and require fewer features to be extracted. Previous work obtains these unbalanced trees by explicitly learning cascade structured classifiers (Viola and Jones, 2004; Dundar and Bi, 2007; Lefakis and Fleuret, 2010; Saberian and Vasconcelos, 2010; Chen et al., 2012; Trapeznikov et al., 2013b; Trapeznikov and Saligrama, 2013a). CSTC can incorporate cascades naturally as a special case, in which the tree of classifiers has only a single node per level of depth. However, further modifications can be made to accommodate the specifics of these settings. We introduce two changes to the learning algorithm:

- Inputs of different classes are re-weighted to account for the severe class imbalance.
- Every classifier node v^k has a terminal element as child and is weighted by the probability of *exiting* rather than the probability of traversing through node v^k.

We refer to the modified algorithm as *Cost-Sensitive Cascade of Classifiers (CSCC)*. An example cascade is illustrated in Figure 4. A CSCC with K-stages is defined by a set of

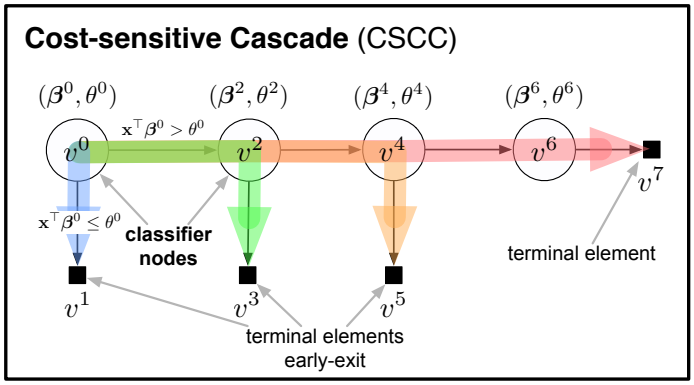

Figure 4: Schematic layout of our classifier cascade with four classifier nodes. All paths are colored in different colors.

weight vectors $\boldsymbol{\beta}^k$ and thresholds θ^k, $\mathcal{C} = \{(\boldsymbol{\beta}^1, \theta^1), (\boldsymbol{\beta}^2, \theta^2), \cdots, (\boldsymbol{\beta}^K, -)\}$. An input is early-exited from the cascade at node v^k if $\mathbf{x}^\top \boldsymbol{\beta}^k < \theta^k$ and is sent to its terminal element v^{k+1}. Otherwise, the input is sent to the next classifier node. At the final node v^K a prediction is made for all remaining inputs via $\mathbf{x}^\top \boldsymbol{\beta}^K$.

In CSTC, most classifier nodes are internal and branch inputs. As such, the predictions need to be similarly accurate for all inputs to ensure that they are passed on to the correct part of the tree. In CSCC, each classifier node early-exits a fraction of its inputs, providing their *final* prediction. As mistakes of such exiting inputs are irreversible, the classifier needs to ensure particularly low error rates for this fraction of inputs. All other inputs are passed down the chain to later nodes. This key insight inspires us to modify the loss function of CSCC from the original CSTC formulation in (6). Instead of weighting the contribution of classifier loss $\ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i)$ by p_i^k, the probability of input $\mathbf{x}_i$ *traversing* through node v^k, we weight it with p_i^{k+1}, the probability of *exiting* through terminal node v^{k+1}. As a second modification, we introduce an optional class-weight $w_{y_i} > 0$ which absorbs some of the impact of the class imbalance. The resulting loss becomes:

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{v^k \in \mathcal{V}} w_{y_i} p_i^{k+1} \ell(\mathbf{x}_i^\top \boldsymbol{\beta}^k, y_i).$$

The cost term is unchanged and the combined cost-sensitive loss function of CSCC becomes

$$\underbrace{\sum_{v^k \in \mathcal{V}} \frac{1}{n} \left(\sum_{i=1}^{n} w_{y_i} p_i^{k+1} \ell_i^k \right) + \rho |\boldsymbol{\beta}^k|}_{\text{regularized loss}} + \underbrace{\lambda \sum_{v^l \in L} p^l \left[\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} (\beta_\alpha^j)^2} \right]}_{\text{feature cost penalty}}. \tag{19}$$

We optimize (19) using the same block coordinate descent optimization described in Section 3.4. Similar as before, we initialize the cascade from left to right, while assuming the currently initialized node is the last node.

4.1 Cronus

CSCC supersedes previous work on cost sensitive learning of cascades by the same authors, Chen et al. (2012). The previous algorithm, named Cronus, shares the same loss terms as CSCC, however the feature and evaluation cost of each node is weighted by the expected number of inputs, p^k, *within* the mixed-norm (highlighted in color):

$$\underbrace{\sum_{v^k \in \mathcal{V}} \frac{1}{n} \left(\sum_{i=1}^{n} w_{y_i} p_i^{k+1} \ell_i^k \right) + \rho |\boldsymbol{\beta}^k|}_{\text{regularized loss}} + \underbrace{\lambda \sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^k \in \mathcal{V}} (p^k \beta_\alpha^k)^2}}_{\text{feature cost penalty}} .$$

In contrast, CSCC in (19) sums over the weighted cost of all exit paths. The two formulations are similar, but CSCC may be considered more principled as it is derived from the exact expected cost of the cascade. As we show in Section 6, this does translate into better empirical accuracy/cost trade-offs.

5. Extension to Non-Linear Classifiers

Although CSTC's decision boundary may be non-linear, each individual node classifier is linear. For many problems this may be too restrictive and insufficient to divide the input space effectively. In order to allow non-linear decision boundaries we map the input into a more expressive feature space with the "boosting trick" (Friedman, 2001; Chapelle et al., 2011), prior to our optimization. In particular, we first train gradient boosted regression trees with a squared loss penalty for T iterations and obtain a classifier $H'(\mathbf{x}_i) = \sum_{t=1}^{T} h_t(\mathbf{x}_i)$, where each function $h_t(\cdot)$ is a limited-depth CART tree (Breiman, 1984). We then define the mapping $\phi(\mathbf{x}_i) = [h_1(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)]^\top$ and apply it to all inputs. The boosting trick is particularly well suited for our feature cost sensitive setting, as each CART tree only uses a small number of features. Nevertheless, this pre-processing step does affect the loss function in two ways: 1. the feature extraction now happens within the CART trees; and 2. the evaluation time of the CART trees needs to be taken into account.

5.1 Feature Cost After the Boosting Trick

After the transformation $\mathbf{x}_i \to \phi(\mathbf{x}_i)$, each input is $T-$dimensional and consequently, we have the weight vectors $\boldsymbol{\beta} \in \mathcal{R}^T$. To incorporate the feature extraction cost into our loss, we define an auxiliary matrix $\mathbf{F} \in \{0,1\}^{d \times T}$ with $F_{\alpha t} = 1$ if and only if the CART tree h_t uses feature f_α. With this notation, we can incorporate the CART-trees into the original feature extraction cost term for a weight vector $\boldsymbol{\beta}$, as stated in (3). The new formulation and its relaxed version, following the mixed-norm relaxation as stated in (11), are then:

$$\sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{t=1}^{T} |F_{\alpha t} \beta_t| \right\|_0 \longrightarrow \sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{t=1}^{T} (F_{\alpha t} \beta_t)^2}.$$

The non-negative sum inside the l_0 norm is non-zero if and only if feature α is used by at least one tree with non-zero weight, *i.e.* , $|\beta_t| > 0$. Similar to a single classifier, we can also

adapt the feature extraction cost of the path through a CSTC tree, originally defined in (8), which becomes:

$$\sum_{\alpha=1}^{d} c_\alpha \left\| \sum_{v^j \in \pi^l} \sum_{t=1}^{T} |F_{\alpha t} \beta_t^j| \right\|_0 \longrightarrow \sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}. \tag{20}$$

5.2 CART Evaluation Cost

The evaluation of a CART tree may be non-trivial or comparable to the cost of feature extraction and its cost must be accounted for. We define a constant $e_t \geq 0$, which captures the cost of the evaluation of the t^{th} CART tree. We can express this evaluation cost for a single classifier with weight vector $\boldsymbol{\beta}$ in terms of the l_0 norm and again apply the mixed norm relaxation (11). The exact (left term) and relaxed evaluation cost penalty (right term) can be stated as follows:

$$\sum_{t=1}^{T} e_t \|\beta_t\|_0 \longrightarrow \sum_{t=1}^{T} e_t |\beta_t|$$

The left term incurs a cost of e_t for each tree h_t if and only if it is assigned a non-zero weight by the classifier, *i.e.* , $\beta_t \neq 0$. Similar to feature values, we assume that CART tree evaluations can be cached and only incur a cost once (the first time they are computed). With this assumption, we can express the exact and relaxed CART evaluation cost along a path π^l in a CSTC tree as

$$\sum_{t=1}^{T} e_t \left\| \sum_{v^j \in \pi^l} |\beta_t^j| \right\|_0 \longrightarrow \sum_{t=1}^{T} e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}. \tag{21}$$

It is worth pointing out, that (21) is analogous to the feature extraction cost with linear classifiers (8) and its relaxation, as stated in (12).

5.3 CSTC and CSCC with Non-Linear Classifiers

We can integrate the two CART tree aware cost terms (20) and (21) into the optimization problem in (12). The final objective of the CSTC tree after the "boosting trick" becomes then

$$\sum_{v^k \in \mathcal{V}} \underbrace{\left(\frac{1}{n} \sum_{i=1}^{n} p_i^k \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \sum_{v^l \in \mathcal{L}} p^l \Bigg[\underbrace{\sum_t e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}}_{\text{CART evaluation cost penalty}} + \underbrace{\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}}_{\text{feature cost penalty}} \Bigg]. \tag{22}$$

The objective in (22) can be optimized with the same block coordinate descent algorithm, as described in Section 3.4. Similarly, the CSCC loss function with non-linear classifiers becomes

$$\underbrace{\sum_{v^k \in \mathcal{V}} \frac{1}{n} \left(\sum_{i=1}^{n} w_{y_i} p_i^{k+1} \ell_i^k \right) + \rho |\boldsymbol{\beta}^k|}_{\text{regularized loss}} + \lambda \sum_{v^l \in L} p^l \Bigg[\underbrace{\sum_t e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}}_{\text{evaluation cost}} + \underbrace{\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}}_{\text{feature cost}} \Bigg].$$

In the same way, Cronus may be adapted for non-linear classification (see: Chen et al., 2012). To avoid over-fitting, we use validation set to perform early-stopping during optimizing objective function 22.

6. Results

In this section, we evaluate CSTC on a synthetic cost-sensitive learning task and compare it with competing algorithms on two large-scale, real world benchmark problems. Additionally, we discuss the differences between our models for several learning settings. We provide further insight by analyzing the features extracted on a these data sets and looking at how CSTC tree partitions the input space. We judge the effect of the cost-sensitive regularization by looking at how removing terms and varying parameters affects CSTC on real world data sets. We also present detailed results of CSTC on a cost-sensitive version of the MNIST data set, demonstrating that it extracts intelligent *per-instance* features. We end by proposing a criterion that is designed to judge if CSTC will perform well on a data set.

6.1 Synthetic Data

We construct a synthetic regression data set consisting of points sampled from the four quadrants of the X, Z-plane, where $X = Z \in [-1, 1]$. The features belong to two categories: cheap features: $sign(x), sign(z)$ with cost $c = 1$, which can be used to identify the quadrant of an input; and expensive features: $z_{++}, z_{+-}, z_{-+}, z_{--}$ with cost $c = 10$, which equal the exact label of an input if it is from the corresponding quadrant (or a random number otherwise). Since in this synthetic data set we do not transform the feature space, we have $\phi(\mathbf{x}) = \mathbf{x}$, and $\mathbf{F}$ (the weak learner feature-usage variable) is the 6×6 identity matrix. By design, a perfect classifier can use the two cheap features to identify the sub-region of an instance and then extract the correct expensive feature to make a perfect prediction. The minimum feature cost of such a perfect classifier is exactly $c = 12$ per instance. We construct the data set to be a regression problem, with labels sampled from Gaussian distributions with quadrant-specific means $\mu_{++}, \mu_{-+}, \mu_{+-}, \mu_{--}$ and variance 1. The individual values for the label means are picked to satisfy the CSTC assumption, *i.e.* , that the prediction of similar labels requires similar features. In particular, as can be seen in Figure 5 (top left), label means from quadrants with negative $z-$coordinates (μ_{+-}, μ_{--}) are higher than those with positive $z-$coordinates (μ_{++}, μ_{-+}).

Figure 5 shows the raw data (top left) and a CSTC tree trained on this data with its predictions of test inputs made by each node. The semi-transparent gray hyperplane shows the values of thresholds, θ, and vertical gray lines show the difference between predicted label and true label, for each instance. In general, in every path along the tree, the first two classifiers split on the two cheap features and identify the correct sub-region of the input. The leaf classifiers extract a single expensive feature to predict the labels. As such, the mean squared error of the training and testing data both approach 0 (and the gray lines vanish) at optimal cost $c = 12$.

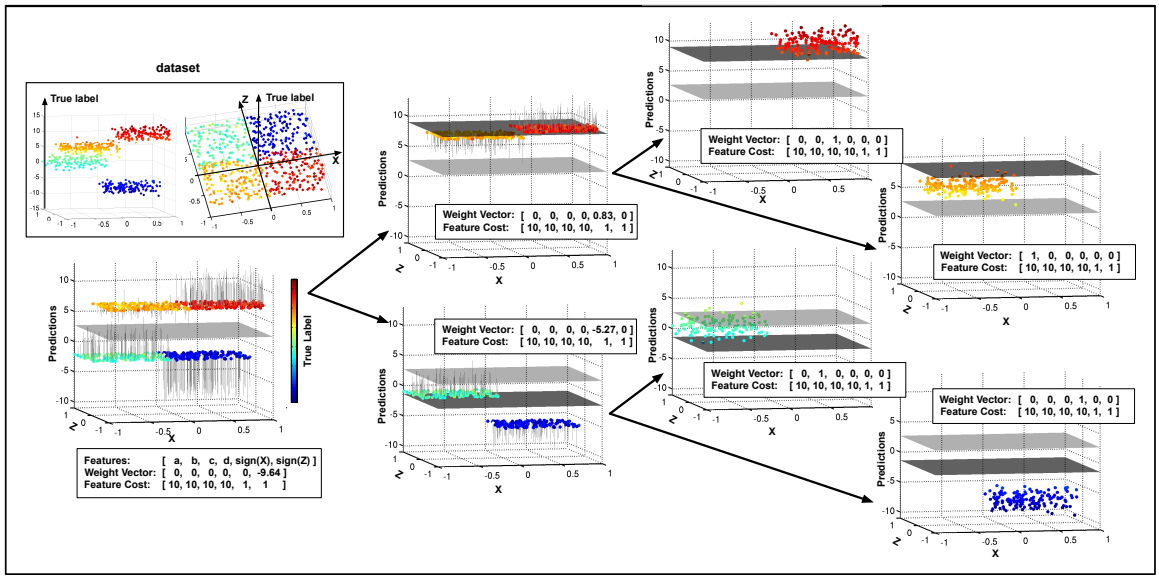

Figure 5: CSTC on synthetic data. The box at left describes the data set. The rest of the figure shows the trained CSTC tree. At each node we show a plot of the predictions made by that classifier and the feature weight vector. The tree obtains a perfect (0%) test-error at the optimal cost of 12 units.

6.2 Yahoo! Learning to Rank

To evaluate the performance of CSTC on real-world tasks, we test it on the Yahoo! Learning to Rank Challenge (LTR) data set. The set contains 19,944 queries and 473,134 documents. Each query-document pair $\mathbf{x}_i$ consists of 519 features. An extraction cost, which takes on a value in the set $\{1, 5, 20, 50, 100, 150, 200\}$, is associated with each feature.[1] The unit of these values turns out to be approximately the number of weak learner evaluations $h_t(\cdot)$ that can be performed while the feature is being extracted. The label $y_i \in \{4, 3, 2, 1, 0\}$ denotes the relevancy of a document to its corresponding query, with 4 indicating a perfect match. We measure the performance using normalized discounted cumulative gain at the 5^{th} position (NDCG@5) (Järvelin and Kekäläinen, 2002), a preferred ranking metric when multiple levels of relevance are available. Let π be an ordering of all inputs associated with a particular query ($\pi(r)$ is the index of the r^{th} ranked document and $y_{\pi(r)}$ is its relevance label), then the NDCG of π at position P is defined as

$$NDCG@P(\pi) = \frac{DCG@P(\pi)}{DCG@P(\pi^*)} \text{ with } DCG@P(\pi) = \sum_{r=1}^{P} \frac{2^{y_{\pi(r)}} - 1}{\log_2(r+1)},$$

where π^* is an optimal ranking (*i.e.* , documents are sorted in decreasing order of relevance). To introduce non-linearity, we transform the input features into a non-linear feature space $\mathbf{x} \to \phi(\mathbf{x})$ with the boosting trick (see Section 5) with $T = 3000$ iterations of gradient boosting and CART trees of maximum depth 4. Unless otherwise stated, we determine the CSTC depth by validation performance (with a maximum depth of 10).

1. The extraction costs were provided by a Yahoo! employee.

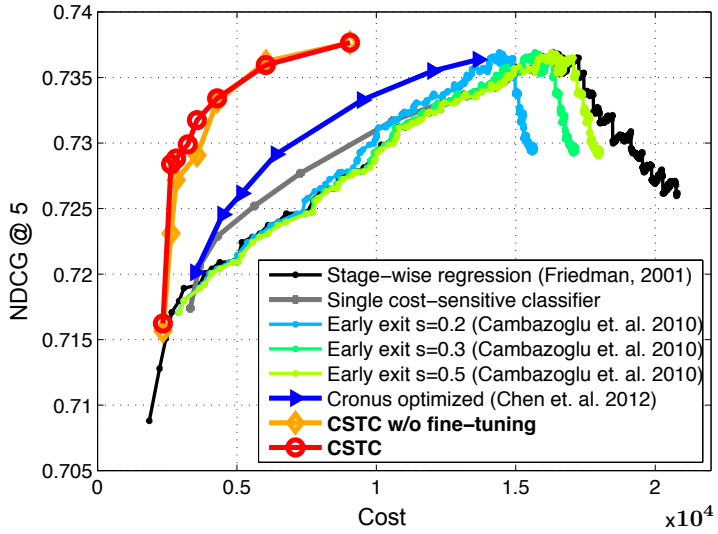

Figure 6: The test ranking accuracy (NDCG@5) and cost of various cost-sensitive classifiers. CSTC maintains its high retrieval accuracy significantly longer as the cost-budget is reduced.

Figure 6 shows a comparison of CSTC with several recent algorithms for test-time budgeted learning. We show NDCG versus cost (in units of weak learner evaluations). We obtain the curves of CSTC by varying the accuracy/cost trade-off parameter λ (and perform early stopping based on the validation data, for fine-tuning). For CSTC we evaluate eight settings, $\lambda = \{\frac{1}{3}, \frac{1}{2}, 1, 2, 3, 4, 5, 6\}$. In the case of *stage-wise regression*, which is not cost-sensitive, the curve is simply a function of boosting iterations. We include CSTC with and without fine-tuning. The comparison shows that there is a small but consistent benefit to fine-tuning the weights as described in Section 3.5.

For competing algorithms, we include *Early exit* (Cambazoglu et al., 2010) which improves upon stage-wise regression by short-circuiting the evaluation of unpromising documents at test-time, reducing the overall test-time cost. The authors propose several criteria for rejecting inputs early and we use the best-performing method "early exits using proximity threshold", where at the i^{th} early-exit, we remove all test-inputs that have a score that is at least $\frac{300-i}{299}s$ lower than the fifth best input, and s determines the power of the early-exit. The *single cost-sensitive classifier* is a trivial CSTC tree consisting of only the root node *i.e.* , a cost-sensitive classifier without the tree structure. We also include Cronus, which is described in Section 4. We set the maximum number of Cronus nodes to 10, and set all other parameters (*e.g.*, keep ratio, discount, early-stopping) based on a validation set. As shown in the graph, both Cronus and CSTC improve the cost/accuracy trade-off curve over all other algorithms. The power of Early exit is limited in this case as the test-time cost is dominated by feature extraction, rather than the evaluation cost. Compared with Cronus, CSTC has the ability to identify features that are most beneficial to different groups of inputs. It is this ability, which allows CSTC to maintain the high NDCG significantly longer

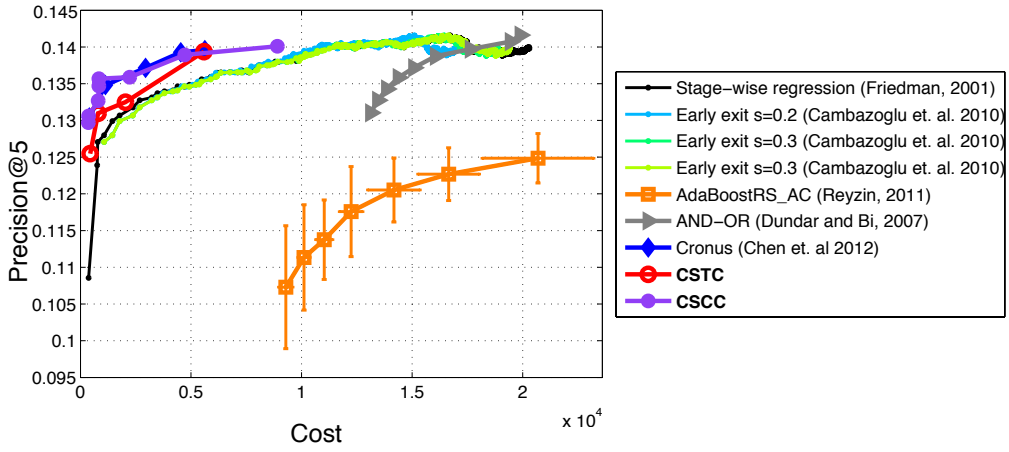

Figure 7: The test ranking accuracy (Precision@5) and cost of various budgeted cascade classifiers on the Skew-LTR data set with high class imbalance. CSCC outperforms similar techniques, requiring less cost to achieve the same performance.

as the cost-budget is reduced. It is interesting to observe that the single cost-sensitive classifier outperforms stage-wise regression (due to the cost sensitive regularization) but obtains much worse cost/accuracy trade offs than the full CSTC tree. This demonstrates that the tree structure is indeed an important part of the high cost effectiveness of CSTC.

6.3 Yahoo! Learning to Rank: Skewed, Binary

To evaluate the performance of our cascade approach CSCC, we construct a highly class-skewed binary data set using the Yahoo! LTR data set. We define inputs having labels $y_i \geq 3$ as 'relevant' and label the rest as 'irrelevant', binarizing the data in this way. We also replicate each negative, irrelevant example 10 times to simulate the scenario where only a few documents are highly relevant, out of many candidate documents. After these modifications, the inputs have one of two labels $\{-1, 1\}$, and the ratio of $+1$ to -1 is $1/100$. We call this data set LTR-Skewed. This simulates an important setting, as in many time-sensitive real life applications the class distributions are often very skewed.

For the binary case, we use the ranking metric Precision@5 (the fraction of top 5 documents retrieved that are relevant to a query). It best reflects the capability of a classifier to precisely retrieve a small number of relevant instances within a large set of irrelevant documents. Figure 7 compares CSCC and Cronus with several recent algorithms for binary budgeted learning. We show Precision@5 versus cost (in units of weak learner evaluations). Similar to CSTC, we obtain the curves of CSCC by varying the accuracy/cost trade-off parameter λ. For CSCC we evaluate eight settings, $\lambda = \{\frac{1}{3}, \frac{1}{2}, 1, 2, 3, 4, 5, 6\}$.

For competing algorithms, in addition to *Early exit* (Cambazoglu et al., 2010) described above, we also include *AND-OR* proposed by Dundar and Bi (2007), which is designed specifically for binary budgeted learning. They formulate a global optimization of a cas-

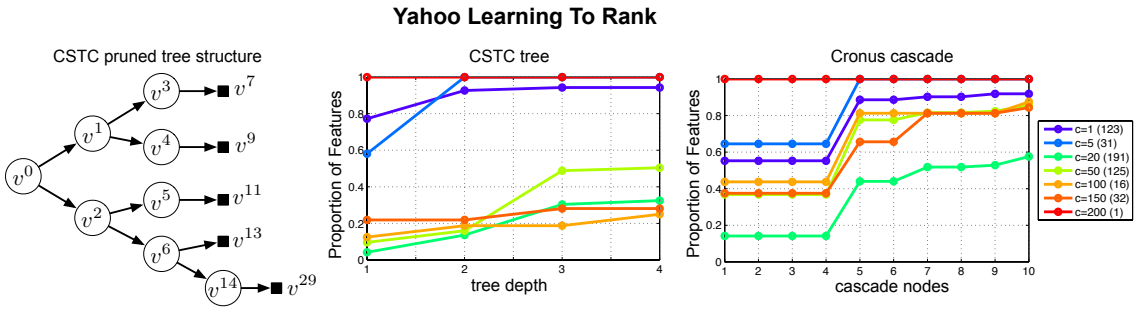

Figure 8: *Left:* The pruned CSTC tree, trained on the Yahoo! LTR data set. The ratio of features, grouped by cost, are shown for CSTC (*center*) and Cronus (*right*). The number of features in each cost group is indicated in parentheses in the legend. More expensive features ($c \geq 20$) are gradually extracted deeper in the structure of each algorithm.

cade of classifiers and employ an *AND-OR* scheme with the loss function that treats negative inputs and positive inputs separately. This setup is based on the insight that positive inputs are carried all the way through the cascade (*i.e.* , each classifier must classify them as positive), whereas negative inputs can be rejected at any time (*i.e.* , it is sufficient if a single classifier classifies them as negative). The loss for positive inputs is the maximum loss across all stages, which corresponds to the AND operation, and encourages all classifiers to make correct predictions. For negative inputs the loss is the minimum loss of all classifiers, which corresponds to the OR operation, and which enforces that at least one classifier makes a correct prediction. Different from our approach, their algorithm requires pre-assigning features to each node. We therefore use five nodes in total, assigning features of cost $\leq 5, \leq 20, \leq 50, \leq 150, \leq 200$. The curve is generated by varying a loss/cost trade-off parameter (similar to λ). Finally, we also compare with the cost sensitive version of AdaboostRS (Reyzin, 2011). This algorithm resamples decision trees, learned with AdaBoost (Freund et al., 1999), inversely proportional to a tree's feature cost. As this algorithm involves random sampling, we averaged over 10 runs and show the standard deviations in both precision and cost.

As shown in the graph, AdaBoostRS obtains lower precision than other algorithms. This may be due to the known sensitivity of AdaBoost towards noisy data, (Melville et al., 2004). AND-OR also under-performs. It requires pre-assigning features prior to training, which makes it impossible to obtain high precision at a low cost. On the other hand, Cronus, CSCC, and CSTC have the ability to cherry pick good but expensive features at an early node, which in turn can reduce the overall cost while improving performance over other algorithms. We take a closer look at this effect in the following section. Cronus and CSCC in general outperform CSTC because they can exit a large portion of the data set early on. As mentioned before, CSCC outperforms Cronus a little bit, which we attribute to the more principled optimization.

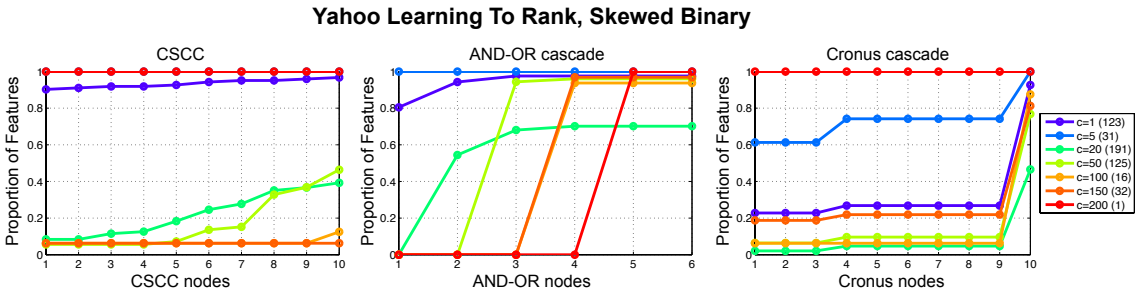

Figure 9: The ratio of features, grouped by cost, that are extracted at different depths of CSCC (*left*), AND-OR (*center*) and Cronus (*right*). The number of features in each cost group is indicated in parentheses in the legend.

6.4 Feature Extraction

Based on the LTR and LTR-Skewed data sets, we investigate the features extracted by various algorithms in each scenario. We fist show the features retrieved in the regular balanced class data set (LTR). Figure 8 (*left*) shows the pruned CSTC tree learned on the LTR data set. The plot in the center demonstrates the fraction of features, with a particular cost, extracted at different depths of the CSTC tree. The rightmost plot shows the features extracted at different nodes of Cronus. We observe a general trend that for both CSTC and Cronus, as depth increases, more features are being used. However, cheap features ($c \leq 5$) are all extracted early-on, whereas expensive features ($c \geq 20$) are extracted by classifiers sitting deeper in the tree. Here, individual classifiers only cope with a small subset of inputs and the expensive features are used to classify these subsets more precisely. The only feature that has cost 200 is extracted at all depths—which seems essential to obtain high NDCG (Chen et al., 2012). Although Cronus has larger depth than CSTC (10 vs 4), most nodes in Cronus are basically dummy nodes (as can be seen by the flat parts of the feature usage curve). For these nodes all weights are zeros, and the threshold is a very small negative number, allowing all inputs to pass through.

In the second scenario, where the class-labels are binarized and are highly skewed (LTR-Skewed), we compare the features extracted by CSCC, Cronus and AND-OR. For a fair comparison, we set the trade-off parameter λ for each algorithm to achieve similar precision 0.135 ± 0.001. We also set the maximum number of nodes of CSCC and Cronus to 10. Figure 9 (*left*) shows the fraction of features, with a particular cost, extracted at different nodes of the CSCC. The center plot illustrates the features used by AND-OR, and the right plot shows the features extracted at different nodes of Cronus. Note that while the features are pre-assigned in the AND-OR algorithm, it still has the ability to only use some of the assigned features at each node. In general, all algorithms use more features as the depth increases. However, compared to AND-OR, both Cronus and CSCC can cherry pick some good but expensive features early-on to achieve high accuracy at a low cost. Some of the expensive features (*e.g.*, $c = 100, 150$) are extracted from the very first node in CSCC and Cronus, whereas in AND-OR, they are only available at the fourth node. This ability is

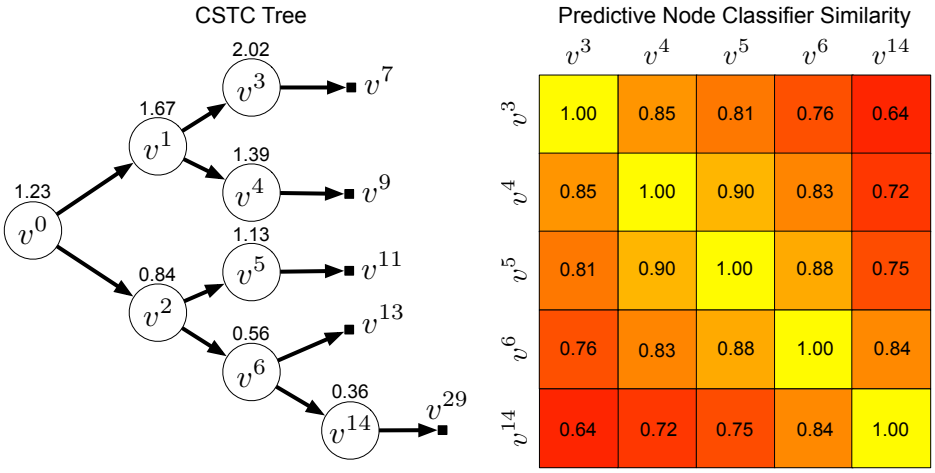

Figure 10: (*Left*) The pruned CSTC-tree generated from the Yahoo! Learning to Rank data set. (*Right*) Jaccard similarity coefficient between classifiers within the learned CSTC tree.

one of the reasons that CSCC and Cronus achieve better performance over existing cascade algorithms.

6.5 Input Space Partition

CSTC has the ability to split the input space and learn more specialized classifiers sitting deeper in the tree. Figure 10 (*left*) shows a pruned CSTC tree ($\lambda = 4$) for the LTR data set. The number above each node indicates the average label of the testing inputs passing through that node. We can observe that different branches aim at different parts of the input domain. In general, the upper branches focus on correctly classifying higher-ranked documents, while the lower branches target low-rank documents. Figure 10 (*right*) shows the Jaccard matrix of the leaf classifiers ($v^3, v^4, v^5, v^6, v^{14}$) from this CSTC tree. The number in field i, j indicates the fraction of shared features between v^i and v^j. The matrix shows a clear trend that the Jaccard coefficients decrease monotonically away from the diagonal. This indicates that classifiers share fewer features in common if their average labels are further apart—the most different classifiers v^3 and v^{14} have only 64% of their features in common—and validates that classifiers in the CSTC tree extract different features in different regions of the tree.

6.6 CSTC Sensitivity

Recall the CSTC objective function with non-linear classifiers,

$$\sum_{v^k \in \mathcal{V}} \underbrace{\left(\frac{1}{n} \sum_{i=1}^{n} p_i^k \ell_i^k + \rho |\boldsymbol{\beta}^k| \right)}_{\text{regularized loss}} + \lambda \sum_{v^l \in \mathcal{L}} p^l \left[\underbrace{\sum_t e_t \sqrt{\sum_{v^j \in \pi^l} (\beta_t^j)^2}}_{\text{CART evaluation cost penalty}} + \underbrace{\sum_{\alpha=1}^{d} c_\alpha \sqrt{\sum_{v^j \in \pi^l} \sum_{t=1}^{T} (F_{\alpha t} \beta_t^j)^2}}_{\text{feature cost penalty}} \right].$$

In order to judge the effect of different terms in the cost-sensitive regularization we experiment with removing the CART evaluation cost penalty (or simply 'evaluation cost') and/or the feature cost penalty. Figure 11 (*Left*) shows the performance of CSTC and the less cost-sensitive variants, after removing one or both of the penalty terms, on the Yahoo! LTR data set. As we suspected, the feature cost term seems to contribute most to the performance of CSTC. Indeed, only taking into account evaluation cost severely impairs the model. Without considering cost, CSTC seems to overfit even though the remaining l_1-regularization prevents the model from extracting all possible features.

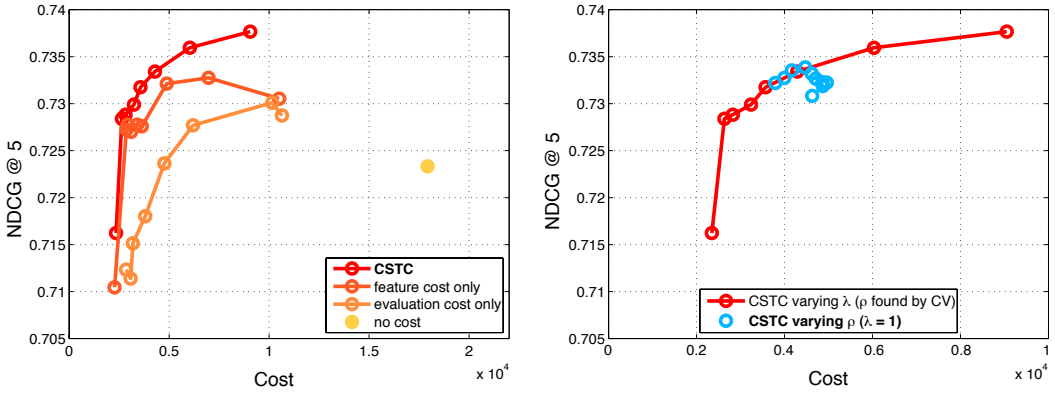

Figure 11: Two plots showing the sensitivity of CSTC to different cost regularization and hyperparameters. *Left:* The test ranking accuracy (NDCG@5) and cost of different CSTC cost-sensitive variants. *Right:* The performance of CSTC for different values of hyperparameter $\rho \in [0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 4, 5]$.

We are also interested in judging the effect of the l_1-regularization hyperparameter ρ on the performance of CSTC. Figure 11 (*Right*) shows for $\lambda = 1$ different settings of $\rho \in [0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 4, 5]$ and the resulting change in cost and NDCG. The result shows that varying ρ does follow the CSTC NDCG/cost trade-off for a little bit, however ultimately leads to a reduction in accuracy. This supports our hypothesis that the cost term is crucial to obtain low cost classifiers.

6.7 Cost-Sensitive MNIST

We created a cost-sensitive binary MNIST data set by first extracting all images of digits 3 and 8. We resized them to four different resolutions: 4×4, 8×8, 16×16, and 28×28 (the

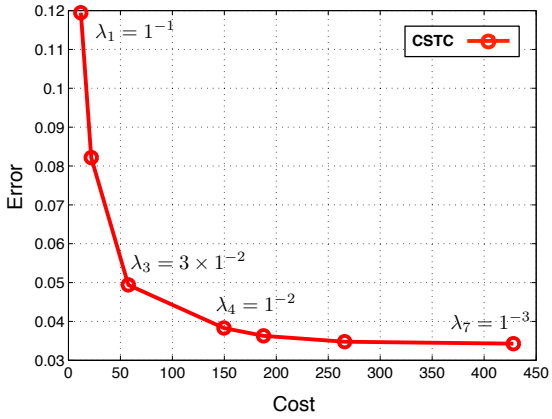

Figure 12: The test error vs. cost trade-off for CSTC on the cost-sensitive MNIST data set

original size), and concatenated all features, resulting in $d = 1120$ features. We assigned each feature a cost $c = 1$. To obtain a baseline error rate for the data set we trained a support vector machine (SVM) with a radial basis function (RBF) kernel. To select hyperparameters C (SVM cost) and γ (kernel width) we used 100 rounds of Bayesian optimization on a validation set (we found $C = 753.1768$, $\gamma = 0.0198$). An RBF-SVM trained with these hyperparameters achieves a test error of 0.005 for cost $c = 1120$. Figure 12 shows error versus cost for different values of the trade-off parameter λ. We note that CSTC smoothly trades off feature cost for error, quickly reducing error initially at small increases in cost.

Figure 13 shows the features extracted and trees built for different values of λ for the cost-sensitive MNIST data set. For each value, we show the paths of one randomly selected 3-instance (lower paths) and one 8-instance (upper paths). For each node in each path we show the features extracted at each of the four resolutions in boxes (red indicates a feature was not extracted). In general, as λ is decreased, more features are extracted. Additionally, for a single λ, nodes along a path tend to use the same features. Finally, even when the algorithm is restricted to use very little cost (*i.e.*, the λ_1 tree) it is still able to find features that distinguish the classes in the data set, sending the 3 and 8-instances along different paths in the tree.

6.8 CSTC Criterion

CSTC implicitly assumes that similarly-labeled inputs can be classified using similar features by sending instances with different predictions to different classifiers. Since not all data sets have such a property, we propose a simple test which indicates if a data set satisfies this assumption. We train binary classifiers with l_1-regularization for neighboring pairs of labels. As an example, the LTR data set contains five labels $\{0, 1, 2, 3, 4\}$, so we train four binary classifiers, (0 *vs.* 1, 1 *vs.* 2, etc.). We then compute the Jaccard coefficient matrix $\mathbf{J}$ between the sparse (because of the l_1-regularizer) feature vectors $\boldsymbol{\beta}$ of all classifiers, where an element $\mathbf{J}_{ij}$ indicates the percentage of overlapping features selected by classifiers i and j. Figure 14 shows this Jaccard matrix on the LTR data set. The figure shows a clear trend that the

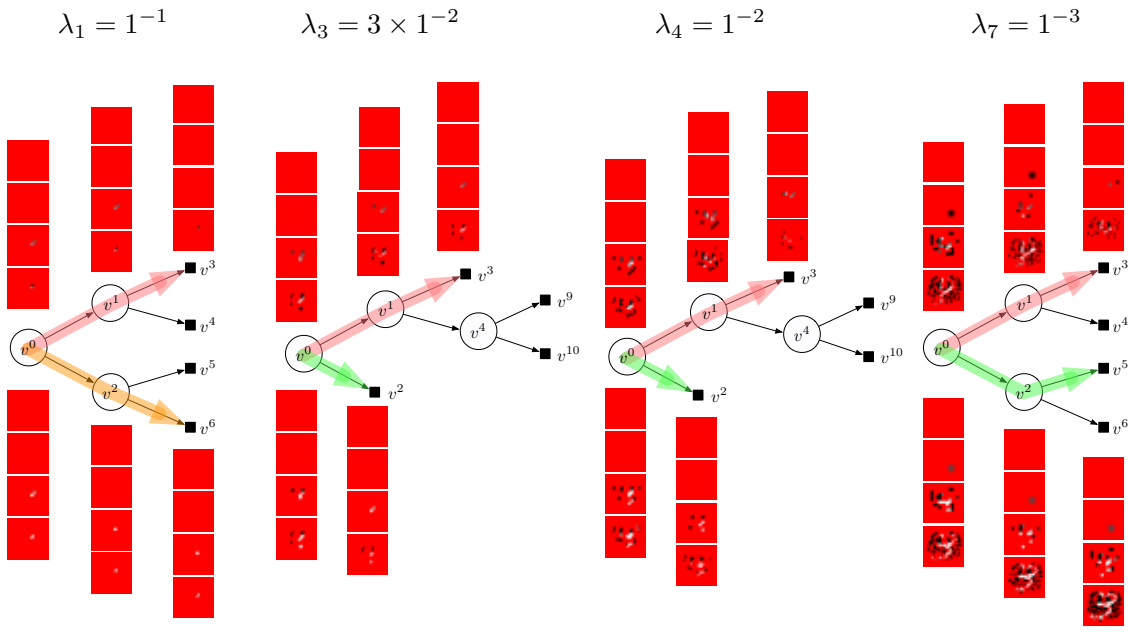

Figure 13: Trees generated for different λ on the cost-sensitive MNIST data set. For each λ we show the path of one 3-instance (orange or green) and one 8-instance (red). For each node these instances visit, we show the features extracted above (for the 8-instance) and below (for the 3-instance) the trees in boxes for different resolutions (red indicates a feature was not extracted).

Jaccard coefficients decrease monotonically away from the diagonal. This indicates that classifiers share fewer features in common if the average label of their training data sets are further apart—a good indication that on this data set CSTC will perform well.

7. Related Work

In the following we review different methods for budgeting the computational cost during test-time starting with simply reducing the feature space via l_1-regularization up to recent work in budgeted learning.

7.1 l_1 Norm Regularization

A related approach to control test-time cost is *feature selection* with l_1 norm regularization (Efron et al., 2004),

$$\min_{\boldsymbol{\beta}} \ell(H(\mathbf{x}; \boldsymbol{\beta}), \mathbf{y}) + \lambda |\boldsymbol{\beta}|,$$

where $\lambda \geq 0$ controls the magnitude of regularization. The l_1-norm regularization results in a sparse feature set (Schölkopf and Smola, 2001), and can significantly reduce the feature cost during test-time (as unused features are never computed). Individual feature cost can be incorporated with feature specific regularization trade-offs, λ_α. The downside of this

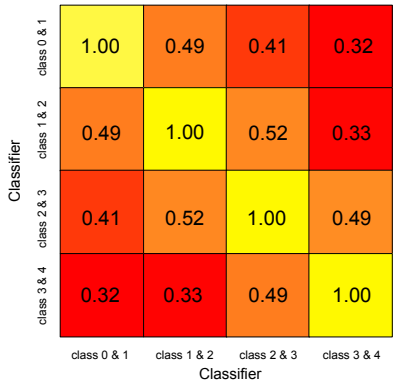

Figure 14: Jaccard similarity coefficient between binary classifiers learned on the LTR data set. The binary classifiers are trained with l_1-regularization for neighboring pairs of labels. As an example, the LTR data set contains five labels $\{0, 1, 2, 3, 4\}$, four binary classifiers are trained, (0 *vs.* 1, 1 *vs.* 2, etc.). There is a clear trend that the Jaccard coefficients decrease monotonically away from the diagonal. This indicates that classifiers share fewer features in common if the average label of their training data sets are further apart—an indication that CSTC will perform well.

approach is that it extracts a feature for *all* inputs or *none*, which makes it uncompetitive with more flexible cascade or tree models.

7.2 Feature Selection

Another approach, extending l_1-regularization, is to select features using some external criterion that naturally limits the number of features used. Cesa-Bianchi et al. (2011) construct a linear classifier given a budget by selecting one instance at a time, then using the current parameters to select a useful feature. This process is repeated until the budget is met. Globerson and Roweis (2006) formulate feature selection as an adversarial game and use minimax to develop a worst-case strategy, assuming feature removal at test-time. These approaches, however, are unaware of the test-time cost in (3), and fail to pick the optimal feature set that best trades-off loss and cost. Dredze et al. (2007) gets closer to directly balancing this trade-off by combining the cost to select a feature with the mutual information of that feature to build a decision tree that reduces the feature extraction cost. This work, though, does not directly minimize the total test-time cost vs. accuracy trade-off of the classifier. Most recently, Xu et al. (2013b) proposed to learn a new feature representation entirely using selected features.

7.3 Linear Cascades

Grubb and Bagnell (2012) and Xu et al. (2012) focus on training a classifier that explicitly trades-off the test-time cost with the loss. Grubb and Bagnell (2012) introduce *SpeedBoost*, a generalization of functional gradient descent for *anytime predictions* (Zilberstein, 1996),

which incorporates the prediction cost during training. The resulting algorithms obtain good approximations at very low cost and refine their predictions if more resources are available. Both algorithms learn classifier cascades that schedule the computational budget and can terminate prematurely if easy inputs reach confident predictions early, to save overall CPU budget for more difficult inputs. This schedule is identical for all inputs, whereas CSTC decides to send inputs along different paths within the tree of classifiers to potentially extract fundamentally different features.

Based on the earlier observation that not all inputs require the same amount of computation to obtain a confident prediction, there is much previous work that addresses this by building classifier cascades (mostly for binary classification) (Viola and Jones, 2004; Dundar and Bi, 2007; Lefakis and Fleuret, 2010; Saberian and Vasconcelos, 2010; Pujara et al., 2011; Chen et al., 2012; Reyzin, 2011; Trapeznikov et al., 2013b). They chain several classifiers into a sequence of stages. Each classifier can either early-exit inputs (predicting them), or pass them on to the next stage. This decision is made based on the prediction of each instance. Different from CSCC, these algorithms typically do not take into account feature cost and implement more ad hoc rules to trade-off accuracy and cost.

7.4 Dynamic Feature Selection During Test-Time

For learning tasks with *balanced classes* and *specialized features*, the linear cascade model is less well-suited. Because all inputs follow exactly the same linear path, it cannot capture the scenario in which different subsets of inputs require different expert features. Chai et al. (2004) introduce the value of unextracted features, where the value of a feature is the increase (gain) in expected classification accuracy minus the cost of including that feature. During test-time, each iteration, their algorithm picks the feature that has the highest value and retrains a classifier with the new feature. The algorithm stops when there is no increase in expected classification rate, or all features are included. Because they employ a naive Bayes classifier, retraining incurs very little cost. Similarly, Gao and Koller (2011) use locally weighted regression during test-time to predict the information gain of unknown features.

Most recently, Karayev et al. (2012) use reinforcement learning during test-time to dynamically select object detectors for a particular image. He et al. (2013) use imitation learning to select instance-specific features for graph-based dependency parsing. Our approach shares the same idea that different inputs require different features. However, instead of learning the best feature for each input during test-time, which introduces an additional cost, we learn and fix a tree structure in training. Each branch of the tree only handles a subset of the input space and, as such, the classifiers in a given branch become specialized for those inputs. Moreover, because we learn a fixed tree structure, it has a test-time complexity that is constant with respect to the training set size.

7.5 Budgeted Tree-Structured Classifiers

Concurrently, there has been work (Deng et al., 2011) to speed up the training and evaluation of tree-structured classifiers (specifically label trees: Bengio et al., 2010), by avoiding many binary one-vs-all classifier evaluations. In many real world data sets the test-time cost is largely composed of feature extraction time and so our aim is different from their work.

Another model (Beygelzimer et al., 2009) learns a tree of classifiers online for estimating the conditional probability of an input label. Their aim is also different from ours as they only consider reducing the training time necessary for the estimation problem. Goetschalckx and Driessens also introduces parsimonious linear model tree to control test-time cost. Possibly most similar to our work is Busa-Fekete et al. (2012), who apply a Markov decision process to learn a directed acyclic graph. At each step, they select features for different instances. Although similar in motivation, their algorithmic framework is very different and can be regarded complementary to ours.

It is worth mentioning that, although Hierarchical Mixture of Experts (HME) (Jordan and Jacobs, 1994) also builds tree-structured classifiers, it does not consider reducing the test-time cost and thus results in fundamentally different models. In contrast, we train each classifier with the test-time cost in mind and each test input only traverses a *single* path from the root down to a terminal element, accumulating path-specific costs. In HME, all test inputs traverse all paths and all leaf-classifiers contribute to the final prediction, incurring the same cost for all test inputs.

8. Conclusions

In this paper, we systematically investigate the trade off between test-time CPU cost and accuracy in real-world applications. We formulate this trade off mathematically for a tree of classifiers and relax it into a well-behaved optimization problem. Our algorithm, Cost-Sensitive Tree of Classifiers (CSTC), partitions the input space into sub-regions and identifies the most cost-effective features for each one of these regions—allowing it to match the high accuracy of the state-of-the-art at a small fraction of the cost. This cost function can be minimized while learning the parameters of all classifiers in the tree jointly.

As the use of machine learning algorithms becomes more and more wide spread, addressing the CPU test-time cost becomes a problem of ever increasing importance. CSTC is one solution but there are still many unanswered questions. Future work will investigate learning theoretical guarantees for test-time budgeted learning, worst-case scenarios (in contrast to average cost), other learning frameworks (*e.g.* , SVM classifiers: Cortes and Vapnik, 1995) and the incorporation of hardware architectures constraints (*e.g.* , clusters, GPUs and shared memory machines). We consider the principled approach of CSTC an important step towards the ultimate goal of fully integrating test-time budgets into machine learning.

Acknowledgements

KQW, ZX, and MK are supported by NIH grant U01 1U01NS073457-01 and NSF grants 1149882 and 1137211.

References

S. Bengio, J. Weston, and D. Grangier. Label embedding trees for large multi-class tasks. *Advances in Neural Information Processing Systems*, 23:163–171, 2010.

A. Beygelzimer, J. Langford, Y. Lifshits, G. Sorkin, and A. Strehl. Conditional probability tree estimation analysis and algorithms. In *Conference on Uncertainty in Artificial Intelligence*, pages 51–58, 2009.

S.P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge Univ Press, 2004.

L. Breiman. *Classification and Regression Trees*. Chapman & Hall/CRC, 1984.

R. Busa-Fekete, D. Benbouzid, B. Kégl, et al. Fast classification using sparse decision DAGs. In *International Conference on Machine Learning*, 2012.

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 *Web Search and Data Mining*, pages 411–420, 2010.

N. Cesa-Bianchi, S. Shalev-Shwartz, and O. Shamir. Efficient learning with partially observed attributes. *The Journal of Machine Learning Research*, 12:2857–2878, 2011.

X. Chai, L. Deng, Q. Yang, and C.X. Ling. Test-cost sensitive naive Bayes classification. In *International Conference on Data Mining*, pages 51–58. IEEE, 2004.

O. Chapelle, P. Shivaswamy, S. Vadrevu, K. Weinberger, Y. Zhang, and B. Tseng. Boosted multi-task learning. *Machine Learning*, 85(1):149–173, 2011.

M. Chen, Z. Xu, K. Q. Weinberger, and O. Chapelle. Classifier cascade for minimizing feature evaluation cost. In *International Conference on Artificial Intelligence and Statistics*, 2012.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

J. Deng, S. Satheesh, A.C. Berg, and L. Fei-Fei. Fast and balanced: Efficient label tree learning for large scale object recognition. In *Advances in Neural Information Processing Systems*, 2011.

M. Dredze, R. Gevaryahu, and A. Elias-Bachrach. Learning fast classifiers for image spam. In *Conference on Email and Anti-Spam*, 2007.

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

B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of Statistics*, 32(2):407–499, 2004.

M. Fleck, D. Forsyth, and C. Bregler. Finding naked people. *European Conference on Computer Vision*, pages 593–602, 1996.

Y. Freund, R. Schapire, and N. Abe. A short introduction to boosting. *Japanese Society for Artificial Intelligence*, 14(771-780):1612, 1999.

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

T. Gao and D. Koller. Active classification based on value of classifier. In *Advances in Neural Information Processing Systems*, pages 1062–1070. 2011.

A. Globerson and S. Roweis. Nightmare at test time: robust learning by feature deletion. In *International Conference on Machine Learning*, pages 353–360. ACM, 2006.

R. Goetschalckx and K. Driessens. Parsimonious linear model trees.

A. Grubb and J. A. Bagnell. Speedboost: Anytime prediction with uniform near-optimality. In *International Conference on Artificial Intelligence and Statistics*, 2012.

H. He, H. Daumé III, and J. Eisner. Dynamic feature selection for dependency parsing. In *Conference on Empirical Methods in Natural Language Processing*, 2013.

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

M.I. Jordan and R.A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

S. Karayev, T. Baumgartner, M. Fritz, and T. Darrell. Timely object recognition. In *Advances in Neural Information Processing Systems*, pages 899–907, 2012.

B. Korte and J. Vygen. *Combinatorial Optimization*, volume 21. Springer, 2012.

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. In *Advances in Neural Information Processing Systems*, pages 1315–1323. 2010.

P. Melville, N. Shah, L. Mihalkova, and R.J. Mooney. Experiments on ensembles with missing and noisy data. In *Multiple Classifier Systems*, pages 293–302. Springer, 2004.

J. Pujara, H. Daumé III, and L. Getoor. Using classifier cascades for scalable e-mail classification. In *Conference on Email and Anti-Spam*, 2011.

L. Reyzin. Boosting on a budget: Sampling for feature-efficient prediction. In *International Conference on Machine Learning*, pages 529–536, 2011.

M. Saberian and N. Vasconcelos. Boosting classifier cascades. In J. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R.S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, pages 2047–2055. 2010.

B. Schölkopf and A.J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT press, 2001.

R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

K. Trapeznikov and V. Saligrama. Supervised sequential classification under budget constraints. In *International Conference on Artificial Intelligence and Statistics*, pages 581–589, 2013a.

K. Trapeznikov, V. Saligrama, and D. Castañón. Multi-stage classifier design. *Machine Learning*, 92(2-3):479–502, 2013b.

P. Viola and M.J. Jones. Robust real-time face detection. *International Journal on Computer Vision*, 57(2):137–154, 2004.

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

Z. Xu, K. Weinberger, and O. Chapelle. The greedy miser: Learning under test-time budgets. In *International Conference on Machine Learning*, pages 1175–1182, 2012.

Z. Xu, M.J. Kusner, M. Chen, and K.Q. Weinberger. Cost-sensitive tree of classifiers. In *International Conference on Machine Learning*, pages 133–141. JMLR Workshop and Conference Proceedings, 2013a.

Z. Xu, M.J. Kusner, G. Huang, and K.Q. Weinberger. Anytime representation learning. In *International Conference on Machine Learning*, pages 1076–1084, 2013b.

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*, pages 1697–1704. Cambridge, MA, 2008.

S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73, 1996.