Statistical Techniques in Robotics (16-831, F09) Lecture #26 (Nov. 19th)

Gradient Boosting

Lecturer: Drew Bagnell *Scribe: Eric Whitman*

1 Sparsity Misconceptions

Last time, we said that we should only run Exponentiated Gradient Descent on life if a small number of things are good.

To analyze this claim, we look at the regret bounds:

Gradient Descent

$R \leq |F|_2 |G|_2 \sqrt{T}$

Exponentiated Gradient Descent

$R \leq |F|_1 |G|_\infty \sqrt{T} \sqrt{\log(d)}$

There also exist algorithms that interpolate between these two and have bounds of the form $|F|_p |G|_q$, where p and q are dual norms ($\frac{1}{p} + \frac{1}{q} = 1$).

By comparing the bounds, we can debunk two common misconceptions about sparsity

1.1 Misconception 1

Claim: w^* must be sparse for exp-grad to make sense.

Look at the example of the continuously rebalanaced portfolio. We also have $\sum w = 1$, which gives the L_1 ball. In the best case for the non-exponential version, we evenly distribute the assets ($w_i = w_j$). This gives $|w|_1 = 1$ and $|w|_2 = \frac{1}{\sqrt{d}}$

We also know that $|G|_\infty \geq G_i$, which gives us

$$|G|_2 = \sqrt{\sum_{i=1}^{d} G_i^2} \leq \sqrt{\sum_{i=1}^{d} |G|_\infty^2} = \sqrt{d} |G|_\infty$$

We can now substitute and compare the bounds:

$|G|_\infty \sqrt{\log d}$ vs. $\frac{1}{\sqrt{d}} \sqrt{d} |G|_\infty$

This simplifies to $\sqrt{\log d}$ vs. 1, meaning that even in the worst case in terms of w^*, exponentiated gradient descent can be (depending on the gradient) only worse by a factor of $\sqrt{\log d}$, which is small.

Truth: If weights vary a lot and $|G|_\infty$ is much different from $|G|_2$, then exponentiated-gradient is good.

1.2 Misconception 2

Claim: If there are lots of features, then do exp-grad.

This is false because the difference in volume between the L_1 and the L_2 ball is larger in high dimensions. The result is that $|G|_2 >> |G|_\infty$. An example is that exp-grad is bad for bag-of-words.

2 Greedy Algorithms

2.1 ϵ-Boosting

1. $w = 0$
2. compute l,∇l
3. identify $i^* = \arg\max_i |\nabla l^i|$
4. $w_{i^*} -= \epsilon \nabla l^{i^*}$

Repeat steps 2-4 until convergence.

One possible loss function is the squared error loss:

$$l = \frac{1}{2}\sum_t (y_t - w^\mathsf{T} x_t)^2$$

$$\nabla l = \sum_t (y_t - w^\mathsf{T} x_t)(-x_t)$$

$$\nabla l_i = -\sum_t (y_t - w^\mathsf{T} x_t)(x_t^i)$$

2.2 Orthogonal Matching Pursuit

The name comes from signal processing. This is normally used with a squared error loss function.

1. $w = 0$; active set $= \emptyset$
2. compute l,∇l
3. identify $i^* = \arg\max_i |\nabla l^i|$
4. active set $\cup = i^*$ (add i^* to the active set)
5. minimize over w restricted to active set

Repeat steps 2-5 until convergence.

Squared loss gives orthogonal residuals, which can make this algorithm quite efficient. It is identical to AdaBoost for squared loss. ϵ-boosting is almost always better than AdaBoost.

Greedy methods are actually sparse whereas exp-grad is only pseudo-sparse. This makes greedy methods better computationally. Usually, most of the computational effort is in step 3: find $i^* = \arg\max_i |\nabla l^i|$. For many problems, we have "oracle" access to i^*, meaning there is some trick to get i^* without checking all possible i's.

2.3 "Oracle" Access to i^*

Suppose the actual features on the data, f, are 10 dimensional. Then x might be all small (depth 2) decision trees on f or x might be all hyperplane seperators on f. In either of these cases, x is extremely high dimensional, but we have some other way of finding the best one.

Take the loss function $l = \sum_t (y_t - f(x_t) + \epsilon h(x_t))^2$, where $f(x_t)$ is what we have so far (the sum of the previously added classifiers), and $h(x_t)$ is a new classifier $h(x) : x \Longrightarrow \{-1, 1\}$.

We can then split the part that does not involve h $(y_t - f(x_t))$ into two parts:

$o_t = \text{sign}(y_t - f(x_t)) = $ target output of h $\alpha_t = |y_t - f(x_t)| = $ weight

If we substitute these in to the loss function, drop the ϵ^2 term, and drop the term that does not depend on h, we can see that minimizing the original loss function is equivalent to minimizing:

$\arg\min_h \sum_t \alpha_t o_t h(x_t)$

This is what we normally do when we train classifiers: we give it correct labelings (o_t) and weights (α_t), so we know how to minimize this efficiently.

This method is not quite correct because the classifier actually minimizes some other cost function rather than the squared loss that we want.

2.4 A Different Loss Function

Rather than squared loss, we can use $l = \sum \max(0, 1 - y_t f(x_t))$

$l' = (1 - y_t f(x_t) > 0)?y : 0$(In C programming langauge notation.)

Splitting this into 2 parts gives:

$o_t = \text{sign}(l'_t)$

$\alpha_t = |l'_t|$