Learning to sort: an overview

Eric Wang

March 24, 2018

Here's the problem:

- We have a set of objects, X.
- X is totally ordered by the relation $\prec$.
- We are also given a set of *rankings*. As we mentioned during our first meeting, we can imagine each ranking is an input-output pair $(X^{(i)}, X^{(i)}_{sorted})$, where
  1. $X^{(i)} \subset X$, and
  2. $X^{(i)}_{sorted}$ is $X^{(i)}$, but ordered according to $\prec$.
- Any algorithm that maps this input to this output should probably be invariant to permutations of the input. Consequently, all the algorithms we've seen embed single examples in some other space Y, such that subsets of Y have some kind of natural ordering.
  1. $Y = \mathbb{R}$: Ranknet and its descendants try to learn an embedding $f : X \longrightarrow \mathbb{R}$ that preserves the order of X:
  $$x_i \prec x_j \iff f(x_i) < f(x_j).$$
  2. $Y = \mathbb{R}^n$: The Gumbel-Sinkhorn learning algorithm tries to learn a function $f : X \longrightarrow \mathbb{R}^n$, where n is the number of entries to be ranked. When n of these are taken together, they parameterize a distribution over permutations whose mode is supposed to sort X according to $\prec$.
- The problem, then, is backpropagating information about the relations between elements of X into the training of the embedding f.
- To do this, we need a loss function
- How should this embedding be trained?
- One approach is RankNet, which trains this embedding with stochastic gradient descent on pairs of elements $x_i, x_j \in X$. RankNet assigns probabilistic interpretations to f and penalizes it with cross-entropy loss. The RankNet algorithm works on any model differentiable with respect to its parameters.
- However, a few things are lost when we train with just two elements at a time.
- But the structure of this input is misleading.

1. Any sorting algorithm should be invariant to the order of $X^{(i)}$, so all the information you need for training is contained in $X^{(i)}_{sorted}$.
2. All the information you need for testing is contained in $X^{(i)}$.

Gumbel-Softmax:

- Say you have a dataset $\mathcal{D}$ and a latent variable Z generating that dataset.
- Z's distribution is parameterized by ϕ:

$$z \sim p_\phi(z)$$

- And let $f(z)$ represent the negative log-likelihood function of $p(z, \mathcal{D})$. (Suppose for now that f is fixed.) We want to minimize $\mathbb{E}[f(z)]$ given ϕ. How do we do this?
- $\nabla_\phi \mathbb{E}_{z\sim p_\phi(z)}[f(z)]$ is intractable. Butf if we reparameterize z as $g(\epsilon, \phi)$ where ϵ is a fixed "noise" random variable, we can work wonders!
- $\mathbb{E}_{\epsilon\sim p(\epsilon)}[f(g(\epsilon, \phi))] = \mathbb{E}_{\epsilon\sim p(\epsilon)}[\nabla_\phi f(g(\epsilon, \phi))]$
- We can estimate this with Monte Carlo.
- Okay, cool. Now we have a trick that lets us find the parameters ϕ that maximize the expected log-likelihood of the data. That is, it helps us choose the distribution in our p_ϕ-space that best approximates the true one.
- But now what if $Val(Z) = D$, a discrete subset of $\mathbb{R}$?
- How can we evaluate $\nabla_\phi f(g(\epsilon, \phi))$ when $g : (\phi, \epsilon) \mapsto D$ can't possibly be differentiable?
- Answer: the Gumbel-Softmax trick.
- Gumbel is the random variable that enables the probability distribution to exist.
- Softmax allows us to adjust our distribution, between temperature 0 (no gradients, exactly one-hotted X) and temperature ∞ (a poor approximation to X, but crazy gradients).
- Let $|D| = K$. The probability simplex over D is Δ^{K-1} of K-dimensional vectors whose components are nonnegative and sum to 1.
- Gumbel-softmax transforms $X \in D$, a discrete variable, into $X^\tau \in \Delta^{K-1}$. (τ is the temperature parameter.)
- It is parameterized by the concentration parameters α_i, and is continuous/differentiable in each! So if α is a function of something else, we can still maximize the likelihood given a discrete distribution over D!
- (Why do we care, though? $\max$ is already differentiable in terms of its inputs.)

The architecture

- Ermon's theorem: let $w : \Sigma \longrightarrow \mathbb{R}^+$ and let $w'(\sigma) = \log w(\sigma) + \gamma_\sigma$, where $\gamma \in \mathbb{R}^\Sigma$ is a vector of samples drawn i.i.d. from $\mathrm{Gumbel}(0)$. Then

$$\Pr\left[\forall \ell > k : w'(\sigma^{(1)}) \geq \cdots \geq w'(\sigma^{(k)}) \geq w'(\sigma^{(\ell)})\right]$$

  is equal to

$$\frac{w(\sigma^{(1)})}{Z} \frac{w(\sigma^{(2)})}{Z - w(\sigma(1))} \cdots \frac{w(\sigma^{(k)})}{Z - \sum_{j=1}^{k-1} w(\sigma^{(j)})}.$$

  Of course, Z is the sum of the w.

- Suppose we want to maximize this probability

Sorting networks:

- We want to learn on entire rankings, not just pairs.
- We also want to use the top-k loss function, rather than penalizing inversions uniformly.
- To do this, we can use Gumbel-Sinkhorn to
- We want a differentiable sorting network, so that we can propagate top-k costs back to the scoring algorithm.
- I suspect that the costs parameterized by k are not actually that different from the costs parameterized by n.
- This is because the cost of sorting out the top k elements are close to sorting the entire array.
- A hard lower bound on the number of comparisons to do this is $n - \log \binom{n}{k}$.

Information Retrieval Measures:

- Historically, measures of information retrieval have been flat. For example, letting l_i be the "correct" relevance label of the ith returned entry, the DCG is defined as

$$DCG@T = \sum_{i=1}^{T} \frac{2^{l_i} - 1}{\log(1 + i)}$$

  And the normalized DCG is:

$$NDCG@T = \frac{DCG@T}{\max DCG@T}$$