Lecture 15: Learning to Rank (with GBDTs)

Why *Machine learned relevance*?

- We used to rank with traditional ranking functions using a very small number of features; possible to tune weighting coefficient by hand.
- But modern (web) systems use a great number of features.

Using classification for ad hoc IR

- Collect a training corpus of (q, d, r) triples
    - r: relevance. binary
    - q and d: query and document
        * Represent the two with a feature vector (α, ω): α is cosine similarity, ω is minimum query window size (the shortest text span that includes *all* query words)
- A linear score function is $\text{Score}(d, q) = \text{Score}(\alpha, \omega) = a\alpha + b\omega + c$.
- This problem setup could be extended to other linear classifiers.

An SVM classifier for information retrieval (Nallapati 2004)

- Features are *not* word presence features, but scores like the summed log tf of all query terms
- Unbalanced data is dealt with by *undersampling* nonrelevant documents during training
- Linear kernel normally best or almost as good as quadratic kernel
- At best the results are about equal to Lemur
- Paper claims that it's easy to add more features (?)

Learning to rank?

- Rather than posing adhoc IR as a classification problem, let's consider it instead as an *ordinal regression* problem.
    - Classification: Mapping to an *unordered* set of classes
    - Ordinal regression: Mapping to an *ordered* set of classes
- Advantages:
    - Relations between relevance levels are modeled.
    - Documents don't possess some *absolute* scale of their goodness. Instead, their goodness are judged against other documents in the collection (for a query.)
- Assume a number of categories **C** of relevance exist
    - These are ordered: $c_1 < c_2 < \cdots < c_j$
- Assume training data is available consisting of
    - Documents query pairs (d, q) represented as feature vectors x_i
    - Relevance ranking c_i