Search Engines:
11-442 / 11-642

Carnegie Mellon University

HW4: Learning to Rank

Software Architecture

You could reuse your existing Indri and BM25 implementations to do this assignment, but your software architecture would be complicated and slow. It is faster and simpler to develop new implementations for this assignment - implementations that are designed for reranking tasks. Your software will only be tested with unstructured queries, so much of what made HW1-HW2 complicated (query operators, default beliefs, structured queries) won't occur in this homework.

Your software should be structured more-or-less as follows:

```
// generate training data
while a training query q is available {
  use QryParser.tokenizeQuery to stop & stem the query
  foreach document d in the relevance judgements for training query q {
    create an empty feature vector
    read the PageRank and spam features from the index
    fetch the term vector for d
    calculate other features for <q, d>
  }

  normalize the feature values for query q to [0..1]
  write the feature vectors to file
}

// train
call svm^rank to train a model

// generate testing data for top 100 documents in initial BM25 ranking
while (a test query q is available) {
  run BM25 to create an initial ranking (on body field)

  use QryParser.tokenizeQuery to stop & stem the query
  foreach document d in the top 100 of initial ranking {
    create an empty feature vector
    read the PageRank and spam features from the index
    fetch the term vector for d
    calculate other features for <q, d>
  }

  normalize the feature values for query q to [0..1]
```

```
    write the feature vectors to file
  }

  // re-rank test data
  call svm^rank to produce scores for the test data
  read in the svm^rank scores and re-rank the initial ranking based on the scores
  output re-ranked result into trec_eval format

  }
```

The main advantage of this implementation is its efficiency and simplicity. It does not use inverted lists, or sort and store ranked lists. Its main disadvantage is that you must reimplement BM25 and Indri to work from a term vector data structure (the same data structure used in HW3). You may assume BOW queries (no query operators of any kind), which simplifies implementation considerably. Pseudo code is shown below for a BM25 implementation that uses term vectors.

```
// tokenize the query before calling featureBM25
queryStems = QryParser.tokenizeQuery (query);

featureBM25 (queryStems, docid, field):
  score = 0
  for each stem in <docid, field>
    if stem is a queryStem
       score += BM25 term score for stem
    end
  end

return score
```

For the Indri retrieval model features, if a field does not match any term of a query, the score for the field is 0. This is consistent with what your HW1-HW2 Indri implementation did - documents that have no terms in common with the query were not given a score.

Note: If you try to instantiate a `TermVector` for a document field that does not exist (e.g., an inlink field for a document that has no inlinks), the constructor returns an empty `TermVector`. It is easy to recognize an empty `TermVector`: The `positionsLength` and `stemsLength` methods will return 0 (i.e., the field does not contain anything).

File Formats

You will encounter several intermediate files during this assignment. This section briefly describes the format of these files. **For more information regarding file formats, visit the SVM^rank website.**

Feature Vector File

After you generate the features for each document, you need to output them in a format SVM^rank understands. Here is an example of such a file.

```
2 qid:1 1:1 2:1 3:0 4:0.2 5:0 # clueweb09-en0000-48-24794
2 qid:1 1:0 2:0 3:1 4:0.1 5:1 # clueweb09-en0011-93-16495
1 qid:1 1:0 2:1 3:0 4:0.4 5:0 # clueweb09-en0132-64-99898
0 qid:1 1:0 2:0 3:1 4:0.3 5:0 # clueweb09-en0370-54-24993
1 qid:2 1:0 2:0 3:1 4:0.2 5:0 # clueweb09-en2108-01-81090
2 qid:2 1:1 2:0 3:1 4:0.4 5:0 # clueweb09-en0760-49-43194
1 qid:2 1:0 2:0 3:1 4:0.1 5:0 # clueweb09-en6303-47-69892
0 qid:2 1:0 2:0 3:1 4:0.2 5:0 # clueweb09-en6373-93-83391
```

```
2 qid:3 1:0 2:0 3:1 4:0.1 5:1 # clueweb09-en2049-70-10797
1 qid:3 1:1 2:1 3:0 4:0.3 5:0 # clueweb09-en1703-12-05495
1 qid:3 1:1 2:0 3:0 4:0.4 5:1 # clueweb09-en8278-44-45799
0 qid:3 1:0 2:1 3:1 4:0.5 5:0 # clueweb09-en9313-84-41193
```

The first column is the score or target value of a <q, d> pair. In a training file, use the relevance value obtained for this <q, d> pair from the relevance judgments ("qrels") file. In a test file, this value should be 0.

The second column is the query id.

The next n columns are feature_id:feature_value pairs, where n is the number of features. Feature ids must be integers starting with 1. Features may be integers or floats. Features must be in canonical order (1 first, 18 last).

A '#' character indicates a comment. They '#' character and everything after it until the end of the line is ignored by SVM^{rank}. However, the grading software examines the comment field. Each line **must** end with the external document id.

The sort key is column 2 (ascending numeric order for the query id portion of the field).

Score File

When you run `svm_rank_classify`, it will create a file in the location specified by letor:rerankingFile. This file contains scores, one per line. The scores are listed in the same order as the test data feature file. You should re-rank your documents based on these new scores. Higher scores are better.

Normalizing Feature Values

The most convenient software architecture creates a feature vector, writes it to disk, creates the next feature vector, writes it to disk, and so on. Unfortunately, SVM^{rank} is a little more effective if your features are all normalized to be in the same range, for example [0..1]. The software does not know the range of Indri, BM25, PageRank, and other features in advance, thus it cannot normalize feature vectors incrementally. Instead, it must wait, and do normalization after all feature vectors for that query are created.

The grading software expects that your features are normalized to [0..1].

To normalize a feature (e.g., f_5) for a specific query, identify the maximum and minimum values for that feature, and then do standard [0..1] normalization. For example, the normalized value for feature f_5 (q_3, d_{21}) is:

> (featureValue_f_5 (q_3, d_{21}) - minValue_f_5 (q_3)) / (maxValue_f_5 (q_3) - minValue_f_5 (q_3)).
>
> featureValue_f_5 (q_3, d_{21}): The value of feature f_5 for query q3 and document d_{21}.
> minValue_f_5 (q_3): The minimum value of feature f_5 across all feature vectors for query q_3.
> maxValue_f_5 (q_3): The maximum value of feature f_5 across all feature vectors for query q_3.

If the min and max are the same value, set the feature value to 0.

Thus, each feature is normalized separately and for each query.

How to Run SVM^{rank}

SVM^{rank} is distributed as a commandline utility, which you can download from the SVM^{rank} website. The website also describes how to call svm_rank_learn to train your model and svm_rank_classify to generate reranking scores for your test data.

The tricky part is that you will be required to run SVM^{rank} from within your Java program. The example below shows how to run svm_rank_learn to train the model. Small modifications to the commandline parameters specified in Runtime.getRuntime().exec() will enable your software to call svm_rank_classify.

```
// runs svm_rank_learn from within Java to train the model
// execPath is the location of the svm_rank_learn utility,
// which is specified by letor:svmRankLearnPath in the parameter file.
// FEAT_GEN.c is the value of the letor:c parameter.
Process cmdProc = Runtime.getRuntime().exec(
    new String[] { execPath, "-c", String.valueOf(FEAT_GEN.c), qrelsFeatureOutputFile,
        modelOutputFile });

// The stdout/stderr consuming code MUST be included.
// It prevents the OS from running out of output buffer space and stalling.

// consume stdout and print it out for debugging purposes
BufferedReader stdoutReader = new BufferedReader(
    new InputStreamReader(cmdProc.getInputStream()));
String line;
while ((line = stdoutReader.readLine()) != null) {
  System.out.println(line);
}
// consume stderr and print it for debugging purposes
BufferedReader stderrReader = new BufferedReader(
    new InputStreamReader(cmdProc.getErrorStream()));
while ((line = stderrReader.readLine()) != null) {
  System.out.println(line);
}

// get the return value from the executable. 0 means success, non-zero
// indicates a problem
int retValue = cmdProc.waitFor();
if (retValue != 0) {
  throw new Exception("SVM Rank crashed.");
}
```

FAQ

If you have questions not answered here, see the HW4 FAQ and the Homework Testing FAQ.

Updated on February 26, 2018
Jamie Callan