package com.twitter.search.earlybird.search.relevance.scoring;

import java.io.IOException;

import java.util.List;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.IndexReader;

import org.apache.lucene.search.Explanation;

import com.twitter.common.collections.Pair;

import com.twitter.search.common.constants.thriftjava.ThriftLanguage;

import com.twitter.search.common.features.thrift.ThriftSearchResultFeatures;

import com.twitter.search.common.query.HitAttributeHelper;

import com.twitter.search.common.relevance.features.EarlybirdDocumentFeatures;

import com.twitter.search.common.results.thriftjava.FieldHitAttribution;

import com.twitter.search.common.schema.base.ImmutableSchemaInterface;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.core.earlybird.index.DocIDToTweetIDMapper;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentAtomicReader;

import com.twitter.search.core.earlybird.index.TimeMapper;

import com.twitter.search.earlybird.common.config.EarlybirdConfig;

import com.twitter.search.earlybird.search.relevance.LinearScoringData;

import com.twitter.search.earlybird.thrift.ThriftSearchResultMetadata;

import com.twitter.search.earlybird.thrift.ThriftSearchResultMetadataOptions;

import com.twitter.search.earlybird.thrift.ThriftSearchResultType;

import com.twitter.search.earlybird.thrift.ThriftSearchResultsRelevanceStats;

import com.twitter.search.queryparser.query.Query;

/\*\*

\* Defines a ranking function which computes the score of a document that matches a query.

\*/

public abstract class ScoringFunction {

/\*\*

\* Returned by a {@link #score(int, float)} to indicate that a hit should be scored below all.

\*

\* We have some equality tests like:

\* "if (score == ScoringFunction.SKIP\_HIT) {...}" (DefaultScoringFunction#updateRelevanceStats)

\* We might also have double to float casts.

\*

\* Such castings seem to work with the equality test, but there might corner cases when casting

\* this float value to a double (and back) might not work properly.

\*

\* If possible, we should choose a constant that is not in the valid score range. Then we can

\* turn the float equality tests into Math.abs(...) < EPSILON tests.

\*/

public static final float SKIP\_HIT = -Float.MAX\_VALUE;

private final ImmutableSchemaInterface schema;

// The current doc ID and the reader for the current segment should be private, because we don't

// want sub-classes to incorrectly update them. The doc ID should only be updated by the score()

// and explain() methods, and the reader should only be updated by the setNextReader() method.

private int currentDocID = -1;

protected DocIDToTweetIDMapper tweetIDMapper = null;

protected TimeMapper timeMapper = null;

protected EarlybirdDocumentFeatures documentFeatures;

protected int debugMode = 0;

protected HitAttributeHelper hitAttributeHelper;

protected Query query;

protected FieldHitAttribution fieldHitAttribution;

public ScoringFunction(ImmutableSchemaInterface schema) {

this.schema = Preconditions.checkNotNull(schema);

}

protected ImmutableSchemaInterface getSchema() {

return schema;

}

/\*\*

\* Updates the reader that will be used to retrieve the tweet IDs and creation times associated

\* with scored doc IDs, as well as the values for various CSFs. Should be called every time the

\* searcher starts searching in a new segment.

\*/

public void setNextReader(EarlybirdIndexSegmentAtomicReader reader) throws IOException {

tweetIDMapper = reader.getSegmentData().getDocIDToTweetIDMapper();

timeMapper = reader.getSegmentData().getTimeMapper();

documentFeatures = new EarlybirdDocumentFeatures(reader);

initializeNextSegment(reader);

}

public void setHitAttributeHelperAndQuery(HitAttributeHelper newHitAttributeHelper,

Query parsedQuery) {

this.hitAttributeHelper = newHitAttributeHelper;

this.query = parsedQuery;

}

public void setFieldHitAttribution(FieldHitAttribution fieldHitAttribution) {

this.fieldHitAttribution = fieldHitAttribution;

}

public void setDebugMode(int debugMode) {

this.debugMode = debugMode;

}

/\*\*

\* Allow scoring functions to perform more per-segment-specific setup.

\*/

protected void initializeNextSegment(EarlybirdIndexSegmentAtomicReader reader)

throws IOException {

// Noop by default

}

// Updates the current document ID and advances all NumericDocValues to this doc ID.

private void setCurrentDocID(int currentDocID) throws IOException {

this.currentDocID = currentDocID;

documentFeatures.advance(currentDocID);

}

/\*\*

\* Returns the current doc ID stored in this scoring function.

\*/

public int getCurrentDocID() {

return currentDocID;

}

/\*\*

\* Compute the score for the current hit. This is not expected to be thread safe.

\*

\* @param internalDocID internal id of the matching hit

\* @param luceneQueryScore the score that lucene's text query computed for this hit

\*/

public float score(int internalDocID, float luceneQueryScore) throws IOException {

setCurrentDocID(internalDocID);

return score(luceneQueryScore);

}

/\*\*

\* Compute the score for the current hit. This is not expected to be thread safe.

\*

\* @param luceneQueryScore the score that lucene's text query computed for this hit

\*/

protected abstract float score(float luceneQueryScore) throws IOException;

/\*\* Returns an explanation for the given hit. \*/

public final Explanation explain(IndexReader reader, int internalDocID, float luceneScore)

throws IOException {

setNextReader((EarlybirdIndexSegmentAtomicReader) reader);

setCurrentDocID(internalDocID);

return doExplain(luceneScore);

}

/\*\* Returns an explanation for the current document. \*/

protected abstract Explanation doExplain(float luceneScore) throws IOException;

/\*\*

\* Returns the scoring metadata for the current doc ID.

\*/

public ThriftSearchResultMetadata getResultMetadata(ThriftSearchResultMetadataOptions options)

throws IOException {

ThriftSearchResultMetadata metadata = new ThriftSearchResultMetadata();

metadata.setResultType(ThriftSearchResultType.RELEVANCE);

metadata.setPenguinVersion(EarlybirdConfig.getPenguinVersionByte());

metadata.setLanguage(ThriftLanguage.findByValue(

(int) documentFeatures.getFeatureValue(EarlybirdFieldConstant.LANGUAGE)));

metadata.setSignature(

(int) documentFeatures.getFeatureValue(EarlybirdFieldConstant.TWEET\_SIGNATURE));

metadata.setIsNullcast(documentFeatures.isFlagSet(EarlybirdFieldConstant.IS\_NULLCAST\_FLAG));

return metadata;

}

/\*\*

\* Updates the given ThriftSearchResultsRelevanceStats instance based on the scoring metadata for

\* the current doc ID.

\*/

public abstract void updateRelevanceStats(ThriftSearchResultsRelevanceStats relevanceStats);

/\*\*

\* Score a list of hits. Not thread safe.

\*/

public float[] batchScore(List<BatchHit> hits) throws IOException {

throw new UnsupportedOperationException("This operation (batchScore) is not implemented!");

}

/\*\*

\* Collect the features and CSFs for the current document. Used for scoring and generating the

\* returned metadata.

\*/

public Pair<LinearScoringData, ThriftSearchResultFeatures> collectFeatures(

float luceneQueryScore) throws IOException {

throw new UnsupportedOperationException("This operation (collectFeatures) is not implemented!");

}

/\*\*

\* Implement this function to populate the result metadata based on the given scoring data.

\* Otherwise, this is a no-op.

\*

\* Scoring functions that implement this should also implement getScoringData().

\*/

public void populateResultMetadataBasedOnScoringData(

ThriftSearchResultMetadataOptions options,

ThriftSearchResultMetadata metadata,

LinearScoringData data) throws IOException {

// Make sure that the scoring data passed in is null because getScoringDataForCurrentDocument()

// returns null by default and if a subclass overrides one of these two methods, it should

// override both.

Preconditions.checkState(data == null, "LinearScoringData should be null");

}

/\*\*

\* This should only be called at hit collection time because it relies on the internal doc id.

\*

\* Scoring functions that implement this should also implement the function

\* populateResultMetadataBasedOnScoringData().

\*/

public LinearScoringData getScoringDataForCurrentDocument() {

return null;

}

}