package com.twitter.search.common.relevance.scorers;

import java.util.Map;

import java.util.concurrent.ConcurrentMap;

import com.google.common.base.Preconditions;

import com.google.common.collect.Maps;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common\_internal.text.version.PenguinVersion;

import com.twitter.search.common.metrics.RelevanceStats;

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.relevance.config.TweetProcessingConfig;

import com.twitter.search.common.relevance.entities.TwitterMessage;

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

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

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

/\*\*

\* Compute a text score for TwitterMessage based on its offensiveness,

\* shoutness, length, readability and hashtag properties extracted from

\* tweet text.

\* <p/>

\* Formula:

\* text\_score = offensive\_text\_damping \* offensive\_username\_damping \*

\* Sigma(feature\_score\_weight \* feature\_score)

\* <p/>

\* scored features are: length, readability, shout, entropy, links

\*/

public class TweetTextScorer extends TweetScorer {

private static final Logger LOG = LoggerFactory.getLogger(TweetTextScorer.class);

private static final double DEFAULT\_OFFENSIVE\_TERM\_DAMPING = 0.2d;

private static final double DEFAULT\_OFFENSIVE\_NAME\_DAMPING = 0.2d;

// Sigma of all weights = 1.0d

private static final double DEFAULT\_LENGTH\_WEIGHT = 0.5d;

private static final double DEFAULT\_READABILITY\_WEIGHT = 0.1d;

private static final double DEFAULT\_SHOUT\_WEIGHT = 0.1d;

private static final double DEFAULT\_ENTROPY\_WEIGHT = 0.25d;

private static final double DEFAULT\_LINK\_WEIGHT = 0.05d;

private static final double DEFAULT\_NO\_DAMPING = 1.0d;

// Sigmoid alpha values for normalization

private static final double DEFAULT\_READABILITY\_ALPHA = 0.05d;

private static final double DEFAULT\_ENTROPY\_ALPHA = 0.5d;

private static final double DEFAULT\_LENGTH\_ALPHA = 0.03d;

private static final ConcurrentMap<String, SearchRateCounter> RATE\_COUNTERS =

Maps.newConcurrentMap();

private static final ConcurrentMap<PenguinVersion, Map<Integer, SearchRateCounter>>

SCORE\_HISTOGRAMS = Maps.newConcurrentMap();

private double offensiveTermDamping = DEFAULT\_OFFENSIVE\_TERM\_DAMPING;

private double offensiveNameDamping = DEFAULT\_OFFENSIVE\_NAME\_DAMPING;

private double lengthWeight = DEFAULT\_LENGTH\_WEIGHT;

private double readabilityWeight = DEFAULT\_READABILITY\_WEIGHT;

private double shoutWeight = DEFAULT\_SHOUT\_WEIGHT;

private double entropyWeight = DEFAULT\_ENTROPY\_WEIGHT;

private double linkWeight = DEFAULT\_LINK\_WEIGHT;

private double readabilityAlpha = DEFAULT\_READABILITY\_ALPHA;

private double entropyAlpha = DEFAULT\_ENTROPY\_ALPHA;

private double lengthAlpha = DEFAULT\_LENGTH\_ALPHA;

/\*\* Configure from a config file, validate the configuration. \*/

public TweetTextScorer(String configFile) {

TweetProcessingConfig.init(configFile);

// get dampings

checkWeightRange(offensiveTermDamping = TweetProcessingConfig

.getDouble("offensive\_term\_damping", DEFAULT\_OFFENSIVE\_TERM\_DAMPING));

checkWeightRange(offensiveNameDamping = TweetProcessingConfig

.getDouble("offensive\_name\_damping", DEFAULT\_OFFENSIVE\_NAME\_DAMPING));

// get weights

checkWeightRange(lengthWeight = TweetProcessingConfig

.getDouble("length\_weight", DEFAULT\_LENGTH\_WEIGHT));

checkWeightRange(readabilityWeight = TweetProcessingConfig

.getDouble("readability\_weight", DEFAULT\_READABILITY\_WEIGHT));

checkWeightRange(shoutWeight = TweetProcessingConfig

.getDouble("shout\_weight", DEFAULT\_SHOUT\_WEIGHT));

checkWeightRange(entropyWeight = TweetProcessingConfig

.getDouble("entropy\_weight", DEFAULT\_ENTROPY\_WEIGHT));

checkWeightRange(linkWeight = TweetProcessingConfig

.getDouble("link\_weight", DEFAULT\_LINK\_WEIGHT));

// check sigma of weights

Preconditions.checkArgument(

lengthWeight + readabilityWeight + shoutWeight + entropyWeight + linkWeight == 1.0d);

readabilityAlpha = TweetProcessingConfig

.getDouble("readability\_alpha", DEFAULT\_READABILITY\_ALPHA);

entropyAlpha = TweetProcessingConfig.getDouble("entropy\_alpha", DEFAULT\_ENTROPY\_ALPHA);

lengthAlpha = TweetProcessingConfig.getDouble("length\_alpha", DEFAULT\_LENGTH\_ALPHA);

}

/\*\* Creates a new TweetTextScorer instance. \*/

public TweetTextScorer() {

}

/\*\* Scores the given tweet. \*/

public void scoreTweet(final TwitterMessage tweet) {

Preconditions.checkNotNull(tweet);

for (PenguinVersion penguinVersion : tweet.getSupportedPenguinVersions()) {

TweetFeatures features = Preconditions.checkNotNull(tweet.getTweetFeatures(penguinVersion));

TweetTextFeatures textFeatures = Preconditions.checkNotNull(features.getTweetTextFeatures());

TweetTextQuality textQuality = Preconditions.checkNotNull(features.getTweetTextQuality());

boolean isOffensiveText = textQuality.hasBoolQuality(

TweetTextQuality.BooleanQualityType.OFFENSIVE);

boolean isOffensiveScreenName = textQuality.hasBoolQuality(

TweetTextQuality.BooleanQualityType.OFFENSIVE\_USER);

double shoutScore = DEFAULT\_NO\_DAMPING - textQuality.getShout();

double lengthScore = normalize(textFeatures.getLength(), lengthAlpha);

double readabilityScore = normalize(textQuality.getReadability(), readabilityAlpha);

double entropyScore = normalize(textQuality.getEntropy(), entropyAlpha);

double score = (isOffensiveText ? offensiveTermDamping : DEFAULT\_NO\_DAMPING)

\* (isOffensiveScreenName ? offensiveNameDamping : DEFAULT\_NO\_DAMPING)

\* (lengthWeight \* lengthScore

+ readabilityWeight \* readabilityScore

+ shoutWeight \* shoutScore

+ entropyWeight \* entropyScore

+ linkWeight \* (tweet.getExpandedUrlMapSize() > 0 ? 1 : 0));

// scale to [0, 100] byte

textQuality.setTextScore((byte) (score \* 100));

updateStats(

isOffensiveText,

isOffensiveScreenName,

textFeatures,

score,

getRateCounterStat("num\_offensive\_text\_", penguinVersion),

getRateCounterStat("num\_offensive\_user\_", penguinVersion),

getRateCounterStat("num\_no\_trends\_", penguinVersion),

getRateCounterStat("num\_has\_trends\_", penguinVersion),

getRateCounterStat("num\_too\_many\_trends\_", penguinVersion),

getRateCounterStat("num\_scored\_tweets\_", penguinVersion),

getScoreHistogram(penguinVersion));

if (LOG.isDebugEnabled()) {

LOG.debug(String.format(

"Tweet length [%.2f] weighted length [%.2f], readability [%.2f] "

+ "weighted readability [%.2f], shout [%.2f] weighted shout [%.2f], "

+ "entropy [%.2f], weighted entropy [%.2f], "

+ "score [%.2f], text [%s], penguin version [%s]",

lengthScore,

lengthWeight \* lengthScore,

readabilityScore,

readabilityWeight \* readabilityScore,

shoutScore,

shoutWeight \* shoutScore,

entropyScore,

entropyWeight \* entropyScore,

score,

tweet.getText(),

penguinVersion));

}

}

}

private void updateStats(boolean isOffensiveText,

boolean isOffensiveScreenName,

TweetTextFeatures textFeatures,

double score,

SearchRateCounter offensiveTextCounter,

SearchRateCounter offensiveUserNameCounter,

SearchRateCounter noTrendsCounter,

SearchRateCounter hasTrendsCounter,

SearchRateCounter tooManyTrendsHashtagsCounter,

SearchRateCounter scoredTweets,

Map<Integer, SearchRateCounter> scoreHistogram) {

// set stats

if (isOffensiveText) {

offensiveTextCounter.increment();

}

if (isOffensiveScreenName) {

offensiveUserNameCounter.increment();

}

if (textFeatures.getTrendingTermsSize() == 0) {

noTrendsCounter.increment();

} else {

hasTrendsCounter.increment();

}

if (TwitterMessage.hasMultipleHashtagsOrTrends(textFeatures)) {

tooManyTrendsHashtagsCounter.increment();

}

scoredTweets.increment();

int bucket = (int) Math.floor(score \* 10) \* 10;

scoreHistogram.get(bucket).increment();

}

// normalize the passed in value to smoothed [0, 1.0d] range

private static double normalize(double value, double alpha) {

return 2 \* (1.0d / (1.0d + Math.exp(-(alpha \* value))) - 0.5);

}

// Make sure weight values are within the range of [0.0, 1.0]

private void checkWeightRange(double value) {

Preconditions.checkArgument(value >= 0.0d && value <= 1.0d);

}

private Map<Integer, SearchRateCounter> getScoreHistogram(PenguinVersion penguinVersion) {

Map<Integer, SearchRateCounter> scoreHistogram = SCORE\_HISTOGRAMS.get(penguinVersion);

if (scoreHistogram == null) {

scoreHistogram = Maps.newHashMap();

String statsName = "num\_text\_score\_%d\_%s";

for (int i = 0; i <= 100; i += 10) {

scoreHistogram.put(i, RelevanceStats.exportRate(

String.format(statsName, i, penguinVersion.name().toLowerCase())));

}

scoreHistogram = SCORE\_HISTOGRAMS.putIfAbsent(penguinVersion, scoreHistogram);

if (scoreHistogram == null) {

scoreHistogram = SCORE\_HISTOGRAMS.get(penguinVersion);

}

}

return scoreHistogram;

}

private SearchRateCounter getRateCounterStat(String statPrefix, PenguinVersion penguinVersion) {

String statName = statPrefix + penguinVersion.name().toLowerCase();

SearchRateCounter rateCounter = RATE\_COUNTERS.get(statName);

if (rateCounter == null) {

// Only one RateCounter instance is created for each stat name. So we don't need to worry

// that another thread might've created this instance in the meantime: we can just create/get

// it, and store it in the map.

rateCounter = RelevanceStats.exportRate(statName);

RATE\_COUNTERS.put(statName, rateCounter);

}

return rateCounter;

}

}