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

import java.io.File;

import java.io.IOException;

import java.io.InputStream;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.atomic.AtomicReference;

import com.google.common.base.Joiner;

import com.google.common.base.Preconditions;

import com.google.common.io.ByteSource;

import com.google.common.util.concurrent.ThreadFactoryBuilder;

import org.apache.commons.io.IOUtils;

import org.apache.commons.lang.StringUtils;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.text.language.LocaleUtil;

import com.twitter.common.text.token.TokenizedCharSequence;

import com.twitter.common.text.token.attribute.TokenType;

import com.twitter.common.util.Clock;

import com.twitter.common\_internal.text.pipeline.TwitterNgramGenerator;

import com.twitter.common\_internal.text.topic.BlacklistedTopics;

import com.twitter.common\_internal.text.topic.BlacklistedTopics.FilterMode;

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

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

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

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

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

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

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

import com.twitter.search.common.util.io.periodic.PeriodicFileLoader;

import com.twitter.search.common.util.text.NormalizerHelper;

import com.twitter.search.common.util.text.TokenizerHelper;

/\*\*

\* Determines if tweet text or username contains potentially offensive language.

\*/

public class TweetOffensiveEvaluator extends TweetEvaluator {

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

private static final int MAX\_OFFENSIVE\_TERMS = 2;

private final File filterDirectory;

private static final File DEFAULT\_FILTER\_DIR = new File("");

private static final String ADULT\_TOKEN\_FILE\_NAME = "adult\_tokens.txt";

private static final String OFFENSIVE\_TOPIC\_FILE\_NAME = "offensive\_topics.txt";

private static final String OFFENSIVE\_SUBSTRING\_FILE\_NAME = "offensive\_substrings.txt";

private static final ThreadLocal<TwitterNgramGenerator> NGRAM\_GENERATOR\_HOLDER =

new ThreadLocal<TwitterNgramGenerator>() {

@Override

protected TwitterNgramGenerator initialValue() {

// It'll generate ngrams from TokenizedCharSequence, which contains tokenization results,

// so it doesn't matter which Penguin version to use here.

return new TwitterNgramGenerator.Builder(PenguinVersion.PENGUIN\_6)

.setSize(1, MAX\_OFFENSIVE\_TERMS)

.build();

}

};

private final AtomicReference<BlacklistedTopics> offensiveTopics =

new AtomicReference<>();

private final AtomicReference<BlacklistedTopics> offensiveUsersTopics =

new AtomicReference<>();

private final AtomicReference<ByteSource> adultTokenFileContents = new AtomicReference<>();

private final AtomicReference<ByteSource> offensiveTokenFileContents = new AtomicReference<>();

private final AtomicReference<ByteSource> offensiveSubstringFileContents = new

AtomicReference<>();

private final SearchCounter sensitiveTextCounter =

RelevanceStats.exportLong("num\_sensitive\_text");

public TweetOffensiveEvaluator() {

this(DEFAULT\_FILTER\_DIR);

}

public TweetOffensiveEvaluator(

File filterDirectory

) {

this.filterDirectory = filterDirectory;

adultTokenFileContents.set(BlacklistedTopics.getResource(

BlacklistedTopics.DATA\_PREFIX + ADULT\_TOKEN\_FILE\_NAME));

offensiveTokenFileContents.set(BlacklistedTopics.getResource(

BlacklistedTopics.DATA\_PREFIX + OFFENSIVE\_TOPIC\_FILE\_NAME));

offensiveSubstringFileContents.set(BlacklistedTopics.getResource(

BlacklistedTopics.DATA\_PREFIX + OFFENSIVE\_SUBSTRING\_FILE\_NAME));

try {

rebuildBlacklistedTopics();

} catch (IOException e) {

throw new RuntimeException(e);

}

ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor(

new ThreadFactoryBuilder()

.setNameFormat("offensive-evaluator-blacklist-reloader")

.setDaemon(true)

.build());

initPeriodicFileLoader(adultTokenFileContents, ADULT\_TOKEN\_FILE\_NAME, executor);

initPeriodicFileLoader(offensiveTokenFileContents, OFFENSIVE\_TOPIC\_FILE\_NAME, executor);

initPeriodicFileLoader(offensiveSubstringFileContents, OFFENSIVE\_SUBSTRING\_FILE\_NAME, executor);

}

private void initPeriodicFileLoader(

AtomicReference<ByteSource> byteSource,

String fileName,

ScheduledExecutorService executor) {

File file = new File(filterDirectory, fileName);

try {

PeriodicFileLoader loader = new PeriodicFileLoader(

"offensive-evaluator-" + fileName,

file.getPath(),

executor,

Clock.SYSTEM\_CLOCK) {

@Override

protected void accept(InputStream stream) throws IOException {

byteSource.set(ByteSource.wrap(IOUtils.toByteArray(stream)));

rebuildBlacklistedTopics();

}

};

loader.init();

} catch (Exception e) {

// Not the end of the world if we couldn't load the file, we already loaded the resource.

LOG.error("Could not load offensive topic filter " + fileName + " from ConfigBus", e);

}

}

private void rebuildBlacklistedTopics() throws IOException {

offensiveTopics.set(new BlacklistedTopics.Builder(false)

.loadFilterFromSource(adultTokenFileContents.get(), FilterMode.EXACT)

.loadFilterFromSource(offensiveSubstringFileContents.get(), FilterMode.SUBSTRING)

.build());

offensiveUsersTopics.set(new BlacklistedTopics.Builder(false)

.loadFilterFromSource(offensiveTokenFileContents.get(), FilterMode.EXACT)

.loadFilterFromSource(offensiveSubstringFileContents.get(), FilterMode.SUBSTRING)

.build());

}

@Override

public void evaluate(final TwitterMessage tweet) {

BlacklistedTopics offensiveFilter = this.offensiveTopics.get();

BlacklistedTopics offensiveUsersFilter = this.offensiveUsersTopics.get();

if (offensiveFilter == null || offensiveUsersFilter == null) {

return;

}

if (tweet.isSensitiveContent()) {

sensitiveTextCounter.increment();

}

// Check for user name.

Preconditions.checkState(tweet.getFromUserScreenName().isPresent(),

"Missing from-user screen name");

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

TweetTextQuality textQuality = tweet.getTweetTextQuality(penguinVersion);

if (tweet.isSensitiveContent()) {

textQuality.addBoolQuality(TweetTextQuality.BooleanQualityType.SENSITIVE);

}

// Check if username has an offensive term

if (isUserNameOffensive(

tweet.getFromUserScreenName().get(), offensiveUsersFilter, penguinVersion)) {

SearchRateCounter offensiveUserCounter = RelevanceStats.exportRate(

"num\_offensive\_user\_" + penguinVersion.name().toLowerCase());

offensiveUserCounter.increment();

textQuality.addBoolQuality(TweetTextQuality.BooleanQualityType.OFFENSIVE\_USER);

}

// Check if tweet has an offensive term

if (isTweetOffensive(tweet, offensiveFilter, penguinVersion)) {

SearchRateCounter offensiveTextCounter = RelevanceStats.exportRate(

"num\_offensive\_text\_" + penguinVersion.name().toLowerCase());

offensiveTextCounter.increment();

textQuality.addBoolQuality(TweetTextQuality.BooleanQualityType.OFFENSIVE);

}

}

}

private boolean isUserNameOffensive(String userName,

BlacklistedTopics offensiveUsersFilter,

PenguinVersion penguinVersion) {

String normalizedUserName = NormalizerHelper.normalizeKeepCase(

userName, LocaleUtil.UNKNOWN, penguinVersion);

List<String> termsToCheck = new ArrayList(TokenizerHelper.getSubtokens(normalizedUserName));

termsToCheck.add(normalizedUserName.toLowerCase());

for (String userNameToken : termsToCheck) {

if (!StringUtils.isBlank(userNameToken) && offensiveUsersFilter.filter(userNameToken)) {

return true;

}

}

return false;

}

private boolean isTweetOffensive(final TwitterMessage tweet,

BlacklistedTopics offensiveFilter,

PenguinVersion penguinVersion) {

TweetTextFeatures textFeatures = tweet.getTweetTextFeatures(penguinVersion);

boolean tweetHasOffensiveTerm = false;

// Check for tweet text.

List<TokenizedCharSequence> ngrams =

NGRAM\_GENERATOR\_HOLDER.get().generateNgramsAsTokenizedCharSequence(

textFeatures.getTokenSequence(), tweet.getLocale());

for (TokenizedCharSequence ngram : ngrams) {

// skip URL ngram

if (!ngram.getTokensOf(TokenType.URL).isEmpty()) {

continue;

}

String ngramStr = ngram.toString();

if (!StringUtils.isBlank(ngramStr) && offensiveFilter.filter(ngramStr)) {

tweetHasOffensiveTerm = true;

break;

}

}

// Due to some strangeness in Penguin, we don't get ngrams for tokens around "\n-" or "-\n"

// in the original string, this made us miss some offensive words this way. Here we do another

// pass of check using just the tokens generated by the tokenizer. (See SEARCHQUAL-8907)

if (!tweetHasOffensiveTerm) {

for (String ngramStr : textFeatures.getTokens()) {

// skip URLs

if (ngramStr.startsWith("http://") || ngramStr.startsWith("https://")) {

continue;

}

if (!StringUtils.isBlank(ngramStr) && offensiveFilter.filter(ngramStr)) {

tweetHasOffensiveTerm = true;

break;

}

}

}

if (!tweetHasOffensiveTerm) {

// check for resolved URLs

String resolvedUrlsText =

Joiner.on(" ").skipNulls().join(textFeatures.getResolvedUrlTokens());

List<String> ngramStrs = NGRAM\_GENERATOR\_HOLDER.get().generateNgramsAsString(

resolvedUrlsText, LocaleUtil.UNKNOWN);

for (String ngram : ngramStrs) {

if (!StringUtils.isBlank(ngram) && offensiveFilter.filter(ngram)) {

tweetHasOffensiveTerm = true;

break;

}

}

}

return tweetHasOffensiveTerm;

}

}