SENTIMENT ANALYSIS ON MOVIE REVIEWS WITH LINGPIPE

Project Report

Under the guidance of

YUNIS LONE

Unify Technologies Pvt Ltd

Submitted by

V.Jyothi Naga Durga

Kakinada Institute of Engineering and Technology,

Kakinada

CERTIFICATION

This is to certify that **Ms.V.Jyothi Naga Durga** of B-tech 4th year from Kakinada Institute of Engineering and Technology, Jawaharlal Nehru Technological University, Kakinada successfully submitted her project report on Sentiment Analysis on movie reviews with Lingpipe.

Signature of the guide

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor **Mr.Yunis Lone** for providing their invaluable guidance, comments and suggestions throughout the course of the project. I would specially thank Yunis Sir for constantly motivating me to work harder.

Also I would like to thank **Ms**. **U.Vijaya Bhargavi** and **Ms. T.Durga Bhavani** for their assistance in the project, for their help during the preparation of the sample and for providing me an overview of the project

ABSTRACT

Sentiment analysis and classification is an area of text classification that began around 2001 and has recently been receiving a lot of attention from researchers. Sentiment analysis involves analyzing textual datasets which contain opinions (e.g., social media, blogs, discussion groups, and internet forums) with the objective of classifying the opinions as positive, negative, or neutral. Classification of textual objects according to sentiment is considered a more difficult task than classification of textual objects according to content because opinions in natural language can be expressed in subtle and complex ways containing slang, ambiguity, sarcasm, irony, and idiom.

Sentiment analysis seeks to identify the view-point(s) underlying a text span; an example application is classifying a movie review as “thumbs up” or “thumbs down”. To determine this sentiment polarity, we propose a novel machine-learning method that applies text-categorization techniques to just the subjective portions of the document. This greatly facilitates incorporation of cross-sentence contextual constraints.

CONTENTS

7.2 EXTENDED CLASSIFIER EVALUATION

7.2.1 TRADITIONAL CLASSIFICATION STATISTICS

7.2.2 ONE-VERSUS-ALL EVALUATIONS

7.2.3 MACRO- AND MICRO-AVERAGED RESULTS

1. INTRODUCTION

Sentiment analysis is becoming one of the most profound research areas for prediction and classification. Automated sentiment analysis of text is used in fields where products and services are reviewed by customers and critics. Thus, sentiment analysis becomes important for businesses to draw a general opinion about their products and services. Our analysis helps concerned organizations to find opinions of people about movies from their reviews, if it is positive or negative. One can in turn formulate a public opinion about a movie. Our goal is to calculate the polarity of sentences that we extract from the text of reviews. We will model sentiment from movie reviews and try to find out how this sentiment matches with the success for these movies.

In other words, if a movie review is positive, negative or neutral. But this task can be difficult and tricky. Consider a sentence "the movie interstellar was visually a treat but the story line was terrible". Now one can clearly see how categorizing this sentence as negative, positive or neutral can be difficult. The phrases "visually a treat" and "story line was terrible" can be considered positive and negative respectively but the degree of their 2 'positiveness' and 'negativeness' is somewhat ambiguous. We use a score for common positive and negative words and use this score to calculate the overall sentiment of a sentence.

2. THE STATEMENT OF THE PROBLEM

Text can be categorized in two types based on its properties in terms of text mining: 'subjectivity' and 'polarity'. The focus of our project is to find the polarity of the text which means that we are interested in finding if the sentence is positive or negative. We use machine learning techniques classify such sentences and try to find answers to the following questions:

1. What machine learning techniques are useful for this purpose? Which one out of them performs the best and which techniques are better than the others?
2. What are some of the advantages and disadvantages of traditional machine learning techniques for sentiment analysis?
3. How difficult the task of extracting sentiment from short comments or sentences can be as compared to the traditional topic based text classification?

3. BACKGROUND

Data available on the Internet comes in a variety of different formats. The simplest and most common format is plain textual data. One example of textual data is online reviews. Online reviews are broadly described as either objective or subjective. An objective review tends to contain mostly facts while a subjective review contains mostly opinions. There are two common review formats the restricted review format and the free format. In a restricted review, the reviewer is asked to separately describe the pros and cons, and to write a comprehensive review. In a free format review, the reviewer writes freely without any separation of the pros and cons.

3.1 OVERVIEW OF SENTIMENT ANALYSIS

Sentiment analysis involves classifying opinions in text into categories like "positive" or "negative" often with an implicit category of "neutral". A classic sentiment application would be tracking what bloggers are saying about a brand like Toyota. Sentiment analysis is also called opinion mining or voice of the customer. There are lots of startups in this area and conferences.
This covers assigning sentiment to movie reviews using language models. There are many other approaches to sentiment. One we use fairly often is sentence based sentiment with a logistic regression classifier. For movie reviews we focus on two types of classification problem:
Subjective (opinion) vs. Objective (fact) sentences
Positive (favorable) vs. Negative (unfavorable) movie reviews

3.2 RESOURCES FOR SENTIMENT ANALYSIS

The high-level idea is to use LingPipe's language classification framework to do two classification tasks: separating subjective from objective sentences, and separating positive from negative movie reviews. In the third section, we show how to build a hierarchical classifier by composing these models.

3.2.1 LINGPIPE

LingPipe is tool kit for processing text using computational linguistics. LingPipe is used to do tasks like: Find the names of people, organizations or locations in news.Automatically classify Twitter search results into categories. Suggest correct spellings of queries.

3.2.1.1 ARCHITECTURE

LingPipe's architecture is designed to be efficient, scalable, reusable, and robust. Highlights include:
Java API with source code and unit tests;
Multi-lingual , multi-domain, multi-genre models;
Training with new data for new tasks;
N-best output with statistical confidence estimates;
Online training (learn-a-little, tag-a-little);
Thread-safe models and decoders for concurrent-read exclusive-write (crew) synchronization; and
Character encoding-sensitive i/o.

3.2.1.2 LINGPIPE DISTRIBUTION

LingPipe is distributed under both commercial licenses and under a royalty-free license.
LingPipe may be downloaded with full source from its web site:
http://alias-i.com/lingpipe/
Other than unpacking the gzipped tarball, there is nothing required for installation. Downloading LingPipe is not technically necessary for running the examples in this book because the LingPipe library jar is included with this book's source download.

3.2.2 JAVA STANDARD EDITION

We chose Java as the basis for LingPipe because we felt it provided the best tradeoff among efficiency, usability, portability, and library availability. We will focus on a few aspects of Java that are particularly crucial for processing textual language data, such as character and string representations, input and output streams and character encodings, regular expressions, parsing HTML and XML markup, etc. In explaining LingPipe's design, we will also delve into greater detail on general features of Java such as concurrency, generics, floating point representations, and the collection package.

This book is based on the latest currently supported standard edition of the Java platform (Java SE), which is version 6. You will need the Java development kit (JDK) in order to compile Java programs. A java virtual machine (JVM) is required to execute compiled Java programs. A Java runtime environment (JRE) contains platform-specific support and integration for a JVM and often interfaces to web browsers for applet support. Java is available in 32-bit and 64-bit versions. The 64-bit version is required to allocate JVMs with heaps larger than 1.5 or 2 gigabytes (the exact maximum for 32-bit Java depends on the platform).

3.2.3 APACHE ANT

Ant is an Apache project used for scripting Java builds. Quoting from Ant's home page, "[Ant] is kind of like Make, but without Make's wrinkles". LingPipe is distributed with a complete set of Ant build files. Apache Ant is a Java library and command-line tool that help building software. Ant has three key features that make it well suited for expository purposes. First, it's portable across all the platforms that support Java. Second, it provides clear XML representations for core Java concepts such as classpaths and command-line arguments. Third, invoking a target in Ant directly executes the dependent targets and then all of the commands in the target. Thus we find Ant builds easier to follow than those using the classic Unix build tool Make or its modern incarnation Apache Maven, both of which attempt to resolve dependencies among targets and determine if targets are up to date before executing them.

You only need one of the binary distributions, which will look like apache-ant-version-bin.tar.gz. First, you need to unpack the distribution. We like directory structures with release names, which is how ant unpacks, using top-level directory names like apache-ant-1.8.1. Then, you need to put the bin subdirectory of the top-level directory into the PATH environment variable so that Ant may be executed from the command line. Ant requires the JAVA_HOME environment variable to be set to the path above the bin directory containing the Java executables. Ant's installation instructions suggest setting the ANT_HOME directory in the same way, and then adding but it's not necessary unless you will be scripting calls to Ant. Ant build files may be imported directly into either the Eclipse or NetBeans IDEs (see below for a description of these). You can test whether Ant is installed with

> ant –version

3.2.4 ECLIPSE IDE

Many pepeople prefer to write (and compile and debug) code in an integrated development environment (IDE). IDEs offer advantages such as automatic method, class and constant completion, easily configurable text formatting, stepwise debuggers, and automated tools for code generation and refactoring. LingPipe development may be carried out through an IDE.

Eclipse provides a full range of code checking, auto-completion and code generation, and debugging facilities. Eclipse is an open-source project with a wide range of additional tools available as plugins. It also has modules for languages other than Java, such as C++ and PHP.

4. DATASET

Lillian Lee and Bo Pang have provided annotated slices of movie review data for polarity (both boolean and scalar), and subjectivity. These three datasets are described at:
Movie Review Data Home Page.
This page is a distribution site for movie-review data for use in sentiment-analysis experiments. Available are collections of movie-review documents labeled with respect to their overall *sentiment Polarity* (positive or negative) or *subjective rating* (e.g., "two and a half stars") and sentences labeled with respect to their *subjectivity status* (subjective or objective) or *polarity*.
We will be using the subjectivity and Boolean polarity data:
Polarity dataset consists of 1000 positive, 1000 negative full text movie reviews. Drawn from IMDB's archive of rec.arts.movies.reviews. Heuristic scripts used to extract first review score from text. Polarity dataset available at:
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
Subjectivity dataset consists of 5000 "objective", 5000 "subjective" sentences. Objective from Internet Movie Database (IMDB) plot summaries, subjective from Rotten Tomatoes customer review "snippets". Subjectivity dataset is available at:
http://www.cs.cornell.edu/people/pabo/movie-review-data/rotten_imdb.tar.gz
Each data file should be downloaded and then unpacked. They are distributed in tarred/gzipped format. We will use POLARITY_DIR to refer to the directory where the reviews were unpacked.

5. IMPLEMENTATION AND METHODOLOGY

5.1 POLARITY ANALYSIS

One can consider document-level polarity classification to be just a special (more difficult) case of text Categorization with sentiment rather than topic-based categories. Hence, standard machine-learning Classification techniques , such as support vector machines (SVMs), can be applied to the entire documents themselves, as was done by Pang, Lee. We refer to such classification techniques as *default polarity classifiers* .
Each default document-level polarity classifier is trained and tested on the extracts formed by applying one of the sentence-level subjectivity detectors to reviews in the polarity dataset. The resulting classifier is able to judge whether a whole movie review is essentially positive or negative (as defined by the data set curators).

Running the Polarity classifier:

Assuming the data is in the directory and the sentimentDemo.jar file exists (if the jar doesn't exist, compile it), the demo may be run. Assuming the data is in the directory POLARITY_DIR and the sentimentDemo.jar file exists (if the jar doesn't exist, compile it), the demo may be run. This produces the following output after running for a minute and quarter on my desktop:

```
BASIC POLARITY DEMO
Data Directory=e:\data\pang-lee-polarity\txt_sentoken
Training.
 # Training Cases=1800
 # Training Chars=6989652
Evaluating.
  # Test Cases=200
  # Correct=163
  % Correct=0.815
```

We've set LingPipe up with its default recommended n-gram length, 8, and the resulting classification accuracy is within .014 of the best accuracy reported in Pang, Lee and Vaithyanathan's Lee's 2002 Thumbs up? paper from EMNLP.

In Confidence Intervals, we show how to compute confidence intervals for these results.

Stepping through the code:

The program reads the training directory location from the command line, trains a classifier on the training **data**, then evaluates the classifier on the test data.

Main to run:

As usual our main method constructs an instance using the command line arguments,then runs it.If any error are thrown, it prints their stack traces.

```
public static void main(String[] args) {
    try {
        new PolarityBasic(args).run();
    } catch (Throwable t) {
        System.out.println("Thrown: " + t);
        t.printStackTrace(System.out);
    }
}
```

Constructor to Marshal Arguments:

Also following our standard operating procedure (SOP), the constructor sets up the member variables.

```
File mPolarityDir;
```

```
String[] mCategories;
DynamicLMClassifier<NGramProcessLM> mClassifier;
PolarityBasic(String[] args) {
 mPolarityDir = new File(args[0],"txt_sentoken");
 mCategories = mPolarityDir.list();
 int nGram = 8;
 mClassifier= DynamicLMClassifier.createNGramProcess(mClassifier,nGram);
}
```

First, the directory is just set to be the directory named txt_sentoken relative to the top-level polarity data directory given as the first command-line argument. The category array is initialized using the directory names under txt_sentoken, which in this case are "pos" and "neg". We set the n-gram length to the constant 8; this could obviously be set with a command-line argument if desired. We use the factory to construct a bounded n-gram classifier with the specified categories and n-gram size. Recall that the process models are normalized for a given input length and do not model boundaries of strings differently than other positions.

Training:

The run method simply calls training then evaluation:

```
void run() throws ClassNotFoundException,
IOException {
   train();
   evaluate();
}
```

We consider training in this section and evaluation in the next. Here's the training method without the code to count the number of cases and characters or the code to print results**:**

```
void train() throws IOException {
   for (int i = 0; i < mCategories.length; ++i) {
      String category = mCategories[i];
      Classification classificatio= new Classification(category);
      File dir = new File(mPolarityDir,mCategories[i]);
      File[] trainFiles = dir.listFiles();
      for (int j = 0; j < trainFiles.length; ++j) {
        File trainFile = trainFiles[j];
       if (isTrainingFile(trainFile)) {
           String review =Files.readFromFile(trainFile,"ISO-8859-1");
           Classified<CharSequence> classified
                        = new  Classified<CharSequence>(review,classification);
           mClassifier.handle(classified)
}
```

```
}
}
```

This method runs through the categories, of which there are two in this demo. It then creates a directory using the polarity data directory and the name of the category. This only works for this demo because the data is organized into directories by category. Then, the potential training files are listed and iterated. For each training file, a test is done to see if it is a training file. If it is, then the text is read from the file using the LingPipe utility method Files.readFromFile, and then used to train the classifier for the specified category.

The only mystery is how we determine if a file is a training file. Lee and Pang were generous enough to pre-slice the files into ten equally-sized slices which are distinguished by the third character of the file name. For instance, the file pos/cv362_15341.txt is a positive training instance in block 3, whereas pos/cv532_6522.txt is a positive training instance in block 5. We just decided to train on blocks 0 through 8 and test on block 9, so the method is just.

```
boolean isTrainingFile(File file) {
   return file.getName().charAt(2) != '9';  // test on fold 9
}
```

Evaluation:

The evaluation code follows the same structure as the training code.

```
void evaluate() throws IOException {
   int numTests = 0;
   int numCorrect = 0;
   for (int i = 0; i < mCategories.length; ++i) {
      String category = mCategories[i];
      File file = new File(mPolarityDir,mCategories[i]);
      File[] testFiles = file.listFiles();
      for (int j = 0; j < testFiles.length; ++j) {
         File testFile = testFiles[j];
         if (!isTrainingFile(testFile)) {
            String review
               = Files.readFromFile(testFile,"ISO-8859-1");
            ++numTests;
            Classification classification
               = mClassifier.classify(review);
            String resultCategory
               = classification.bestCategory();
            if (resultCategory.equals(category))
               ++numCorrect;
```

```
        }
      }
    }
}
```

The code rendered in grey is the same as in the training loop described in the last section. The remaining code begins by setting the number of tests and number of correct answer counters to zero. Then, as each review is processed, the number of tests is incremented. Then the classifier is used to produce a classification for a review string in a single line. Next, the classification's best category is extracted as the result category of classification. If the result category matches the test category, the number of correct classifications is incremented.

The final results are then printed with this code:

```
System.out.println("  # Test Cases="+ numTests);
System.out.println("  # Correct="+ numCorrect);
System.out.println("  % Correct="+ ((double)numCorrect)
                                    /(double)numTests);
```

5.2 BASIC SUBJECTIVITY ANALYSIS:

This section covers a second form of sentiment analysis, namely determining if a sentence is "objective" or "subjective" (again, as defined by the database curators). It follows pretty much the same pattern as the last example, with a slightly different data format, the addition of the classifier evaluation framework from com.aliasi.classify, and a step to compile the model to a file for later use. The advantage of the evaluation framework is that it cannot only tell right from wrong, but also distinguish several shades of grey.

Running the Subjectivity Classifier:

Assuming the data is in the directory POLARITY_DIR and the sentimentDemo.jar file exists (if it doesn't, run ant jar to create it), the demo may be run.

This produces the following output after about 45 seconds of chugging away on my desktop:

```
BASIC SUBJECTIVITY DEMO
Data Directory=e:\data\pang-lee-polarity
Training.
# Sentences plot=5000
# Sentences quote=5000
Compiling.
```

```
  Model file=subjectivity.model
  # Training Cases=9000
  # Training Chars=1160539
Evaluating.
CLASSIFIER EVALUATION
Categories=[plot, quote]
Total Count=1000
Total Correct=921
Total Accuracy=0.921
…
```

We'll step through the output (which runs to more than a page) a bite-sized chunk at a time; the ellipses (...) indicate that the report is continued. The first line of the report indicates the name of the categories. In this case, it's plot for "objective" sentences (drawn from plot summaries) and quote for "subjective" sentences (drawn from user-review snippets). Next comes the same accuracy report as we computed by hand in the last demo. This performance is much better at 92% accuracy than the polarity classification results we saw in the last section.

After the basic accuracy report, the confusion matrix is presented in a format to provide easy inclusion into a spreadsheet or other graphing package such as gnuplot.

```
...
Confusion Matrix
reference \ response
  ,plot,quote
  plot,458,42
  quote,37,463
...
```

This matrix represents the count of all reference/response pairs. The reference categories are read down the left and the response categories along the top. For this demo, the reference is the "gold standard" defined by the database curators and the response is the first-best category produced by the classifier we just trained. Reading the results, there are 458 test cases that were classified as plots by the reference and plots by the response. There are 42 cases that were labeled as plots in the gold standard, but were misclassified as quotes by the classifier. On the next row, there are 463 cases that were labeled quotes in the gold standard that were correctly classified as quotes by our classifier. In addition, there were 37 cases labeled as quotes in the gold standard that our classifier mislabeled as plots.

The remainder of the output is discussed in Extended Classifier Evaluation.

Stepping through the Code:

The code for the subjectivity classifier is almost identical to that of the polarity classifier, so we only focus on a few differences in this section.

Splitting the Training File:

Unlike in the polarity demo, where every case was included in a separate file, here every case is on a single line within a file, so the code's a bit differenent.

```
for (int i = 0; i < mCategories.length; ++i) {
   String category = mCategories[i];
   Classification classification
      = new Classification(category);
   File file = new File(mPolarityDir,
                        mCategories[i] + ".tok.gt9.5000");
   String data = Files.readFromFile(file);
   String[] sentences = data.split("\n");
   int numTraining = (sentences.length * 9) / 10;
   for (int j = 0; j < numTraining; ++j) {
      String sentence = sentences[j];
      Classified<CharSequence> classified
         = new Classified<CharSequence>(sentence,classification);
      mClassifier.handle(classified);
   }
}
```

Here, we construct a file using the specified pattern and then read all of the data from the file. We then split on newlines to derive the sentences. The number of training instances is set to 90% of the input data. We then just loop over the training data instances and train as before.

Writing the Model to a File*:*

The remaining code in the train() method simply compiles the model to a file.

```
FileOutputStream fileOut = new FileOutputStream("subjectivity.model");
ObjectOutputStream objOut = new ObjectOutputStream(fileOut);
mClassifier.compileTo(objOut);
objOut.close();
```

This actually transforms the format, with the resulting model being much faster at runtime. A more robust implementation would handle the close in a finally block that made sure the file output stream was closed to ensure no dangling file pointers were held.

Evaluation:

The evaluation code this time is even simpler with the use of the evaluator.

```
void evaluate() throws IOException {
```

```
  BaseClassifierEvaluator<String> evaluator
    = new BaseClassifierEvaluator<String>(mclassifier,mCategories);
  for (int i = 0; i < mCategories.length; ++i) {
    String category = mCategories[i];
    Classification classification = new Classification(category);
    File file = new File(mPolarityDir, mCategories[i]
                  + ".tok.gt9.5000");
    String data = Files.readFromFile(file,"ISO-8859-1");
    String[] sentences = data.split("\n");
    int numTraining = (sentences.length * 9) / 10;
    for (int j = numTraining;
         j < sentences.length; ++j) {
      Classified<CharSequence> classified
        = new Classified<CharSequence>(sentences[j],classification);
      evaluator.handle(classified);
    }
  }
  System.out.println(evaluator.toString());
}
```

The first line simply creates an evaluator from the classifier and the array of categories. The greyed out parts of the code are identical to that in the training method. In particular, note that it calculates the 10 percent of the data on which to test and then as each sentence is encountered, it is added as a case to the evaluator using the BaseClassifierEvaluator.handle(classified). Evaluation cases consist of the reference result, in this case category, and the input, in this case sentences[j]. Because the evaluator has a handle on the classifier, it just runs the classifier over the input and records the results. When the evaluation loop is done, we just call the toString() method on the evaluator to print out the results. Note that the category supplied to the evaluator is only used for evaluation purposes; the classifier will be used to perform classification on the input sentence without reference to the reference category. The resulting scored classification is then added as an evaluation case with the specified reference category for computing results.

5.3 HEIRARCHIAL POLARITY ANALYSIS

Pang and Lee (2004) introduce a hierarchical approach to classification. Specifically, they use the subjectivity classifier to extract subjective sentences from reviews to be used for polarity classification. Hierarchical models are quite common in the classification and general statistics and machine learning literatures.

Running the Hierarchical Classifier:

The hierarchical classifier is run just like the other demos.

```
HIERARCHICAL POLARITY DEMO
Data Directory=E:\data\pang-lee-polarity\txt_sentoken
Reading Compiled Model
Training.
  # Training Cases=1800
  # Training Chars=6989652
CLASSIFIER EVALUATION
Categories=[neg, pos]
Total Count=200
Total Correct=170
Total Accuracy=0.85
Confusion Matrix
reference \ response
  ,neg,pos
  neg,82,18
  pos,12,88
...
```

Inspecting the Code:

The source for this demo is identical to the first demo in the way that it steps through data, so we don't repeat that code here.

Construction: Reading in the Model

The constructor in this implementation does all of the work of the one in the basic polarity demo in setting up member variables (indicated by ellipses (...)). It also reads in the subjectivity model from a hardwired file named subjectivity.model:

```
...
Classifier<CharSequence,JointClassification> mSubjectivityClassifier;
PolarityHierarchical(String[] args)
    throws ClassNotFoundException, IOException {
    ...
    File modelFile = new File("subjectivity.model");
    System.out.println("\nReading model from file="
                       + modelFile);
    FileInputStream fileIn
        = new FileInputStream(modelFile);
    ObjectInputStream objIn
        = new ObjectInputStream(fileIn);
    mSubjectivityClassifier
        = (JointClassifier<CharSequence>) objIn.readObject();
    objIn.close();
```

```
}
```

This code raises the possibility of either an IOException from the I/O or a ClassNotFoundException from reading in the actual classifier from the object intput stream.

Training:

The next step is to train the polarity classifier. It does this in exactly the same way as in the basic polarity demo using the method train(). This brings up the possibility of only training on the subjective sentences of the training data, but when we did this, we found it hurt rather than helped performance, so we do not include that technique here. This does illustrate another aspect of the nearly limitless fiddling that's possible with these kinds of models.

Evaluation:

The evaluation is set up similarly to the subjectivity demo with a slight twist:

```
void evaluate() throws IOException {
    BaseClassifierEvaluator<CharSequence> evaluator
      = new BaseClassifierEvaluator<CharSequence>(null,mCategories,false);
   for (int i = 0; i < mCategories.length; ++i) {
      String category = mCategories[i];
      File file = new File(mPolarityDir,mCategories[i]);
      File[] trainFiles = file.listFiles();
      for (int j = 0; j < trainFiles.length; ++j) {
         File trainFile = trainFiles[j];
         if (!isTrainingFile(trainFile)) {
            String review
               = Files.readFromFile(trainFile);
            String subjReview
               = subjectiveSentences(review);
            Classification classification
               = mClassifier.classify(subjReview);
            evaluator.addClassification(category,
                                        classification);
         }
      }
   }
   System.out.println();
   System.out.println(evaluator.toString());
}
```

As in previous examples, the code that remains the same is greyed out. The new code creates a classifier evaluator with a null classifier. This is because we don't actually have an implementation of the BaseClassifier interface (see the next section for an illustration of how to create one). Instead, we'll

create classifications on our own and add them to the evaluation. This is illustrated in the remaining new code. Here, a string subjReview is the result of applying the method subjectiveSentences to the review. This extracts the subjective sentences and returns them as a string. We then create a classification using the filtered intput subjReview. Finally, we add it as a case to the evaluator. This illustrates how the evaluator may be used without an embedded classifier -- cases are just added in terms of the first-best answer and the response classification.

The meat of this implementation is in pulling out the subjective sentences. There are many ways this can be configured to run. We implement a technique that reduces a review to 5 to 25 sentences. This will be the five most subjective sentences as ranked by conditional probability of the subjectivity model, as well as up to 20 more sentences if they are 50% or more likely to be subjective according to the subjectivity mode.

```
static int MIN_SENTS = 5;
static int MAX_SENTS = 25;
String subjectiveSentences(String review) {
   String[] sentences = review.split("\n");
   BoundedPriorityQueue<ScoredObject<String>> pQueue
      = new BoundedPriorityQueue<ScoredObject<String>>(ScoredObject.comparator(),
                                          MAX_SENTS);
   for (int i = 0; i < sentences.length; ++i) {
      String sentence = sentences[i];
      ConditionalClassification subjClassification = (ConditionalClassification)
         mSubjectivityClassifier.classify(sentences[i]);
      double subjProb;
      if (subjClassification.category(0).equals("quote"))
         subjProb = subjClassification.conditionalProbability(0);
       else
         subjProb = subjClassification .conditionalProbability(1);
      pQueue.add(new ScoredObject(sentence,subjProb));
   }StringBuilder reviewBuf = new StringBuilder();
   Iterator<ScoredObject<String>> it = pQueue.iterator();
   for (int i = 0; it.hasNext(); ++i) {
      ScoredObject<String> so = it.next();
      if (so.score() < 0.5 && i >= MIN_SENTS) break;
      reviewBuf.append(so.getObject() + "\n");
   }
   String result = reviewBuf.toString().trim();
   return result;
}
```

The first line simply breaks the review into sentences. We then create a priority queue of objects ordered by score with a maximum size of the maximum number of sentences returned. Then we just iterate over

the sentences and classify them with the subjectivity classifier we created in the constructor. The result is cast to a conditional classification, which allows us to extract conditional probabilities. The probability that the sentence is subjective is then set into the variable subjProb. This is a bit tricky because we have to determine if the category quote (meaning "subjective") is the first-best or second-best response and pull out the correct probability. We then add a scored object to the queue with its object set to be the current sentence and its score the conditional probability of that sentence being subjective. The priority queue keeps the items in ranked order up to the specified maximum.

The next batch of code creates a buffer into which to append the subjective sentences. We create an iterator over the elements in the priority queue, which returns the item in order of highest estimated probability of being subjective. If we already have five sentences and the probability of being subjective is less than half (0.5), we break out of the loop. Otherwise, we append the next sentence. Finally, we return the result after trimming any residual whitespace (probably just the final newline; not a very efficient way to do this at all, but these data sets are miniscule).

6. CLASSIFIERS AND THEIR IMPLEMENTATION

6.1 WHAT IS CLASSIFIER?

A classifier takes inputs, which could be just about anything, and return a classification of the input over a finite number of discrete categories. For example, a classifier might take a biomedical research paper's abstract and determine if it is about genomics or not. The outputs are "about genomics" and "not about genomics." Classifiers can have more than two outputs. For instance, a classifier might take a newswire article and classify whether it is politics, sports, entertainment, science and technology, or world or local news; this is what Google and Bing's news services do.1 Other applications of text classifiers, either alone or in concert with other components, range from classifying documents by topic, identifying the language of a snippet of text, analyzing whether a movie review is positive or negative, linking a mention of a gene name in a text to a database entry for that gene, or resolving the sense of an ambiguous word like bank as meaning a savings institution or the sloped land next to a stream (it can also mean many other things, including the tilt of an airplane, or even an adjective for a kind of shot in billiards). In each of these cases, we have a known finite set of outcomes and some evidence in the form of text.

6.2 BUILDING A JOINTCLASSIFIER IMPLEMENTATION

In order to play nicely with the rest of LingPipe, we should really define our hierarchical classifier to implement the interface classify. Classifier. This is actually very straightforward, but we didn't want to confuse the earlier code with tricky Java particulars like inner class interface implementations.
We could either create a class, or we can just create one inline, as if we were javax.swing programmers writing GUI event handlers:

```
JointClassifier<CharSequence> hierClassifier
   = new JointClassifier<CharSequence>() {
      public JointClassification classify(CharSequence input) {
         String review = input.toString();
         String subjReview
            = subjectiveSentences(review);
         return mClassifier.classify(subjReview);
      }
};
```

We could also build a class that reads both models in from a file, or takes both classifiers as parameters, or any number of other solutions in-between. By constructing a hierarchical classifier in any of these ways, it may be supplied to an evaluator.

6.3 PLUG-AND-PLAY CLASSIFIERS

Although we've used 8-gram character language model classifiers, it's easy to plug-and-play any of LingPipe's other classifiers.

Modifying the Existing Classes:

Naive Bayes can be evaluated by importing the relevant classes and setting the classifier in the constructor:

```
import com.aliasi.classify.NaiveBayesClassifier;
import com.aliasi.tokenizer.IndoEuropeanTokenizerFactory;
...
PolarityHierarchical() {
...
TokenizerFactory factory= new IndoEuropeanTokenizerFactory();
int charNGramLength = 0;
 int nuchals = 128;
 mClassifier = new NaiveBayesClassifier(mCategories, factory,charNGramLength, nuchals);
...
```

Run this way, the subjectivity classifier still uses character 8-grams, but the polarity classifier uses naive Bayes with no character language model smoothing. The number of characters (here set to 128), determines the amount of penalty per character for unknown words.

It's also possible to experiment with token n-grams. And, of course, each of these classifiers has a few parameters to tweak. Each model may also be pruned, for further control. Just don't be fooled by overtrained *a posteriori* results on a test set. You're unlikely to be called out on this in an *ACL* paper, but the results are rarely achievable in practice.

We'll end with some words of warning on efficiency and performance. If you use naive Bayes or token n-gram models, you should probably compile them first. Uncompiled naive Bayes takes almost ten minutes to complete the classification demo. The act of compiling them to a file actually precomputes almost all of the probabilities; reading them back in computes the suffix tree backoffs. The compiled classifiers should work in exactly the same way as the classifier from which they were compiled.

The second word of warning is that naive Bayes as implemented here isn't accurate for this task. Typically, naive Bayes as used in classifiers is smoothed using something like add-one (Laplace) smoothing. This is what Pang and Lee do for their naive Bayes baseline. The way to implement add-one smoothing over LingPipe's naive Bayes implementation is to collect all of the tokens during the first training pass in a set. Then they may be added as training data for both categories. This approach may also be helpful for the basic character-level models, but they're usually more robust with respect to smoothing than token models, which is just another reason why we prefer them for most application.

6.4 IMPLEMENTATION OF BASECLASSIFIEREVALUATOR

Class BaseClassifierEvaluator<E>
java.lang.Object
com.aliasi.classify.BaseClassifierEvaluator<E>

Type Parameters:

E - The type of objects being classified by the evaluated classifier.

All Implemented Interfaces:

Handler, ObjectHandler<Classified<E>>

Direct Known Subclasses:

RankedClassifierEvaluator

BaseClassifierEvaluator:

public class **BaseClassifierEvaluator<E>**
extends Object
implements ObjectHandler<Classified<E>>

A BaseClassifierEvaluator provides an evaluation harness for first-best classifiers. An evaluator is constructed from a classifier and a complete list of the categories returned by the classifier.
Test cases are then added using the handle(Classified) which accepts a string-based category and object to classify. The evaluator will run the classifier over the input object and collect results over multiple cases.
There are subtypes of this classifier that evaluate richer classifiers.
An exhaustive set of evaluation metrics for first-best classification results is accessbile as a confusion matrix through the confusionMatrix() method. Confusion matrices provide dozens of statistics on classification which can be computed from first-best results; see ConfusionMatrix for more information.

Thread Safety

This class requires concurrent read and synchronous write synchronization. Reads are any of the statistics gathering methods and write is just adding new test cases.

Storing Cases

This class always stores the classification results and true category of an cases. There is a flag in the constructor that additionally allows the inputs for cases to be stored as part of an evaluation. If the flag is set to true, all input cases are stored. This enables the output of true positives, false positives, false negatives, and true negatives through the methods of the same names

7. CROSS-VALIDATING FACTORY IMPLEMENTATION

Cross-validation performs multiple divisions of the data into training and test sets and then averages the results in order to bring down evaluation variance in order to tighten confidence intervals.

7.1 CONFIDENCE INTERVALS

As usual, we compute 95% confidence intervals for a given accuracy of p over N trials using the binomial distribution bionmial(p,N), which is just the distribution corresponding to N independent trials each with a p chance of success. The deviation of the binomial distribution is:

```
dev(binomial(p,N)) = sqrt(p*(1-p)/N)
```

Consider the basic polarity evaluation, for which accuracy is 81.5%, or 0.815 over 200 cases. This leads to a deviation of

```
dev(binomial(0.815,200))
= sqrt(0.815 * (1.0 - 0.815)/200
```

```
= 0.0275
```

This is a huge deviation, primarily because there are only 200 test cases. A 95% confidence interval is roughly plus or minus 1.6 deviations, or about +/- 0.044. In interval terms, we're 95% confident our true performance is in the interval (77.1, 85.9). In layman's terms, we're not particularly confident about our results for the basic polarity evaluation. If we had 2000 tests rather than 200 (ten times as much data), that number would be +/- 0.014, a factor of sqrt(10) less due to 10 times the amount of data.

We have a much tighter bound for our basic subjectivity demo. There there are 1000 test cases and a 92.1% accuracy, leading to

```
dev(binomial(0.921,1000))
= sqrt(0.815 * (1.0 - 0.815)/200
= 0.00853
```

So for those results, our 95% confidence interval is the narrower (90.7, 93.5).

Pang and Lee used paired t-tests over cross-validated slices of data for significance, but we can't use that tighter technique to compare our results to theirs because we don't have access to their results for the requisite pairing.

7.2 EXTENDED CLASSIFIER EVALUATION

This appendix dives more deeply into the statistical analysis of the results.

7.2.1 TRADITIONAL CLASSIFICATION STATISTICS

The report continues with a range of standard statistics that have been applied to classification problems:

```
...
Random Accuracy=0.5
Random Accuracy Unbiased=0.5000125
kappa=0.8420000000000001
kappa Unbiased=0.8419960499012477
kappa No Prevalence =0.8420000000000001
...
```

The most popular statistic here is the kappa statistic, which we present in all three forms: "standard", adjusted for "bias", and adjusted for "prevalence". See the class documentation for more information about these statistics. Suffice it to say here that this kappa value is well within the rule-of-thumb range expected for "reliable" classification.

The next basic statistics report on information-theoretic measures about the marginal and conditional distributions produced by the training data and the results

```
...
Reference Entropy=1.0
Response Entropy=0.9999278640456615
Cross Entropy=1.0000721383590225
Joint Entropy=1.3983977819792184
Conditional Entropy=0.39839778197921816
Mutual Information=0.6015300820664435
Kullback-Liebler Divergence=7.213835902255758E-5
...
```

The conditional entropy and mutual information statistics are the most informative. The conditional entropy statistic tells us how many additional bits we'd need on average to encode the classification result given the reference category. This number will be 0.0 if there is perfect classification. The mutual information statistic just presents the response entropy minus the conditional entropy.

These are then followed by some statistical measures, including a chi-squared independence test (Pearson's C_2 statistic) and a few-other fairly widely used statistics, which are explained in the class documentation

```
...
chi Squared=709.034903490349
chi-Squared Degrees of Freedom=1
phi Squared=0.709034903490349
Cramer's V=0.8420421031577632
lambda A=0.842
lambda B=0.8404040404040404
...
```

The final bit of this section of the report includes results about average performance from a ranked, scored, conditional and joint probability perspective. The average reference rank indicates the average position on the n-best list provided by a ranked classifier of the gold standard answer. The average score of the reference is just that -- the average score returned by the scored classifier for the reference category; note that because language model classification uses a joint probability classification scheme, the log2 joint probability of the reference is the same as the score (although it is expressed as a cross-entropy rate). The average conditional probabilty says that on average, the classifier was only 55.8% confident in its answer.

```
...
Average Reference Rank=0.079
Average Score Reference=-1.9029423566865242
Average Conditional Probability Reference=0.5575955442436352
```

```
Average Log2 Joint Probability Reference=-1.9029423566865242
...
```

7.2.1 ONE-VERSUS-ALL EVALUATIONS

Next up are one-versus-all evaluations for each category. These indicate performance on a category-by-category basis. This isn't so interesting in our case, because both categories perform about equally well, which is typical in two-category problems with roughly balanced false positives and false negatives.
A one-versus-all evaluation is created by reducing an n-way confusion matrix to a two-way confusion matrix between a given category and everything else. (Like the other statistics, this is thoroughly explained in the class documentation.) Even for a two-way classification problem, these reports provide some interesting insight into the classificatioin problem. Here's the initial piece of the report for the category of objective sentences, plot:

```
...
ONE VERSUS ALL EVALUATIONS BY CATEGORY
CATEGORY[0]=plot
First-Best Precision/Recall Evaluation
  Total=1000
  True Positive=458
  False Negative=42
  False Positive=37
  True Negative=463
  Positive Reference=500
  Positive Response=495
  Negative Reference=500
  Negative Response=505
  Accuracy=0.921
  Recall=0.916
  Precision=0.9252525252525252
  Rejection Recall=0.926
  Rejection Precision=0.9168316831683169
  F(1)=0.9206030150753768
  Fowlkes-Mallows=497.49371855331003
  Jaccard Coefficient=0.8528864059590316
  Yule's Q=0.9854499831466986
  Yule's Y=0.8422896535427215
  Reference Likelihood=0.5
  Response Likelihood=0.495
```

```
  Random Accuracy=0.5
  Random Accuracy Unbiased=0.5000125
  kappa=0.8420000000000001
  kappa Unbiased=0.8419960499012477
  kappa No Prevalence=0.8420000000000001
  chi Squared=709.034903490349
  phi Squared=0.709034903490349
  Accuracy Deviation=0.008529888627643386
...
```

This is just a confusion matrix report based on the number of true positives, false negatives, false positives and true negatives. For the plot category, recall was slightly lower than precision.

The one-versus-all report for the plot category continues with histograms of rank, average rank, conditional and joint probabilities, just as before, but broken out one-versus-all:

```
...
Rank Histogram=
  plot,quote
  458,42
Average Rank Histogram=
  plot,quote
  0.084,0.916
Average Score Histogram=
  plot,quote
  -1.9181814022265598,-2.256328242824601
Average Conditional Probability Histogram=
  plot,quote
  0.5579020517702641,0.442097948229736
Average Joint Probability Histogram=
  plot,quote
  -1.9181814022265598,-2.256328242824601
...
```

The report concludes with two scored precision recall evaluations. These are the kinds of reports produced for information retrieval tasks as used, for example, in the Text Retrieval Conference (TREC).

```
...
Scored One Versus All
  Area Under PR Curve (interpolated)=0.7906839673018342
  Area Under PR Curve (uninterpolated)=0.7878864243381857
  Area Under ROC Curve (interpolated)=0.7840919999999995
  Area Under ROC Curve (uninterpolated)=0.7840920000000003
  Average Precision=0.7878864243381849
  Maximum F(1) Measure=0.7386569872958257
```

```
  BEP (Precision-Recall break even point)=0.7069943289224953
...
```

These include cumulative statistics from the interpolated and uninterpolated precision-recall (PR) and receiver operating characteristic (ROC) curves determined as described in the scored precision-recall evaluation documentation. After the area under the curves, there is average precision, which is very similar, but only includes points which were correct in the average. The maximum F(1)-measure indicates the best possible operating point achievable by setting a threshold. The BEP indicates the best score possible when precision is equal to recall.

Note that the first set of results compares cases using their score, which is just their joint log probabilities (divided by the number of characters, and thus expressed as negative cross-entropy rates). These results are not very good. What this means is that the joint probabilities assigned to cases perform very well at ranking plots versus quotes (92% accuracy), but not very good at ranking confidence overall. For instance, in one case, there might be a score of -1.2 for plot and -1.4 for quote, whereas in another there might be a score of -1.9 for plot and -2.1 for quote. Ranking these provides case 1 plot, then case 1 quote, then case 2 plot then case 2 quote. Even if both decisions were right (both were indeed plots), the ranked scores suffer.

Usually, the conditional one-versus-all scores will be much better. These use the conditional probabilities assigned to answers to rank output. This is likely to be the better approach to setting overall thresholds in one-versus all cases, because the scores are true conditional probabilities after the traditional Bayesian normalization.

```
...
Conditional One Versus All
  Area Under PR Curve (interpolated)=0.9796868652411705
  Area Under PR Curve (uninterpolated)=0.9792351050063459
  Area Under ROC Curve (interpolated)=0.9786800000000003
  Area Under ROC Curve (uninterpolated)=0.9786799999999973
  Average Precision=0.9792351050063466
  Maximum F(1) Measure=0.9216867469879518
  BEP (Precision-Recall break even point)=0.9126984126984127
...
```

The average precision being much higher than the accuracy tells us that we are doing a good job ranking by confidence in that we tend to be more correct when we are more confident.

7.2.3 MACRO- AND MICRO-AVERAGED RESULTS

The final section of the report provides macro- and micro-averaged results. These averages are over the one-versus-all results, which are broken out category-by-category at the end of the report (see below).

The thing to remember is that the macro-averaged results average the one-versus-all results with each category weighted equally

```
...
Macro-averaged Precision=0.9210421042104211
Macro-averaged Recall=0.921
Macro-averaged F=0.9209980249506238
...
```

In general, these can diverge widely from the accuracy figures if either the test data or classification results are skewed toward more populous categories (as is often the case with unbalanced training data). The micro-averaged results weight by case, not by category, treating all cases as equal. This is calculated by summing the one-versus-all matrices and presenting the result as a precision-recall evaluation. As noted in the classifier evaluation documentation and again in the report, micro-averaging leads to multiplying the number of cases by the number of categories and it results in a number of symmetries in counts.

```
...
Micro-averaged Results
        the following symmetries are expected:
          TP=TN, FN=FP
          PosRef=PosResp=NegRef=NegResp
          Acc=Prec=Rec=F
  Total=2000
  True Positive=921
  False Negative=79
  False Positive=79
  True Negative=921
  Positive Reference=1000
  Positive Response=1000
  Negative Reference=1000
  Negative Response=1000
  Accuracy=0.921
  Recall=0.921
  Precision=0.921
  Rejection Recall=0.921
  Rejection Precision=0.921
  F(1)=0.9209999999999999
  Fowlkes-Mallows=1000.0
  Jaccard Coefficient=0.8535681186283596
  Yule's Q=0.9853923195573459
  Yule's Y=0.842
  Reference Likelihood=0.5
```

```
Response Likelihood=0.5
Random Accuracy=0.5
Random Accuracy Unbiased=0.5
kappa=0.8420000000000001
kappa Unbiased=0.8420000000000001
kappa No Prevalence=0.8420000000000001
chi Squared=1417.928
phi Squared=0.708964
Accuracy Deviation=0.006031542091372652
```

Basically, this is just another precision-recall evaluation over aggregate data.