Understanding information content with Apache Tika

Skill Level: Intermediate

Oleg Tikhonov
Software Engineer
IBM

Chris Mattmann (chris.a.mattmann@jpl.nasa.gov)
Senior Computer Scientist, Professor
NASA JPL/USC

15 Jun 2010

With the increasingly widespread use of computers and the pervasiveness the modern Internet has attained, huge amounts of information in many languages are becoming available. Automatic information processing and retrieval is urgently needed to understand content across cultures, languages, and continents. A recent Apache software project, Tika, is becoming an important tool toward realizing content understanding.

Section 1. Introduction

In this tutorial, we introduce the Apache Tika framework and explain its concepts (e.g., N-gram, parsing, mime detection, and content analysis) via illustrative examples that should be applicable to not only seasoned software developers but to beginners to content analysis and programming as well. We assume you have a working knowledge of the Java™ programming language and plenty of content to analyze.

Throughout this tutorial, you will learn:

- Apache Tika's API, most relevant modules, and related functions

- Apache Nutch (one of the progenitors of Tika) and its NgramProfiler and LanguageIdentifier classes, which have recently been ported to Tika
- cpdetector, the code page detector project, and its functionality

What is Apache Tika?

As Apache Tika's site suggests, Apache Tika is a toolkit for detecting and extracting metadata and structured text content from various documents using existing parser libraries.

The parser interface

The org.apache.tika.parser.Parser interface is the key component of Apache Tika. It hides the complexity of different file formats and parsing libraries while providing a simple and powerful mechanism for client applications to extract structured text content and metadata from all sorts of documents. All this is achieved with a single method:

```
void parse(InputStream stream, ContentHandler handler, Metadata metadata)
    throws IOException, SAXException, TikaException;
```

The `parse` method takes the document to be parsed and related metadata as input, and outputs the results as XHTML SAX events and extra metadata. The main criteria that led to this design are shown in Table 1.

Table 1. Criteria for Tika parsing design

Criterion	Explanation
`Streamed parsing`	The interface should require neither the client application nor the parser implementation to keep the full document content in memory or spooled to disk. This allows even huge documents to be parsed without excessive resource requirements.
`Structured content`	A parser implementation should be able to include structural information (headings, links, etc.) in the extracted content. A client application can use this information, for example, to better judge the relevance of different parts of the parsed document.
`Input metadata`	A client application should be able to include metadata like the file name or declared content type with the document to be parsed. The parser implementation can use this information to better guide the parsing process.
`Output metadata`	A parser implementation should be able to return

	document metadata in addition to document content. Many document formats contain metadata, such as the name of the author, that may be useful to client applications.

These criteria are reflected in the arguments of the `parse` method.

Document InputStream

The first argument is an `InputStream` for reading the document to be parsed.

If this document stream cannot be read, parsing stops and the thrown `IOException` is passed up to the client application. If the stream can be read but not parsed (if the document is corrupted, for example), the parser throws a `TikaException`.

The parser implementation will consume this stream, but will not close it. Closing the stream is the responsibility of the client application that opened it initially. Listing 1 shows the recommended pattern for using streams with the `parse` method.

Listing 1. Recommended pattern for using streams with the parse method

```
InputStream stream = ...;      // open the stream
try {
    parser.parse(stream, ...); // parse the stream
} finally {
    stream.close();            // close the stream
}
```

XHTML SAX events

The parsed content of the document stream is returned to the client application as a sequence of XHTML SAX events. XHTML is used to express structured content of the document, and SAX events enable streamed processing. Note that the XHTML format is used here only to convey structural information, not to render the documents for browsing.

The XHTML SAX events produced by the parser implementation are sent to a `ContentHandler` instance given to the `parse` method. If the content handler fails to process an event, parsing stops and the thrown `SAXException` is passed up to the client application.

Listing 2 shows the overall structure of the generated event stream (with indenting added for clarity).

Listing 2. Structure of the generated event stream

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>...</title>
  </head>
  <body>
    ...
  </body>
</html>
```

Parser implementations typically use the `XHTMLContentHandler` utility class to generate the XHTML output. Dealing with the raw SAX events can be complex, so Apache Tika (since V0.2) comes with several utility classes that can be used to process and convert the event stream to other representations.

For example, the `BodyContentHandler` class can be used to extract just the body part of the XHTML output and feed it as SAX events to another content handler or as characters to an output stream, a writer, or simply a string. The following code snippet parses a document from the standard input stream and outputs the extracted text content to standard output:

```
ContentHandler handler = new BodyContentHandler(System.out);
parser.parse(System.in, handler, ...);
```

Another useful class is `ParsingReader` that uses a background thread to parse the document and returns the extracted text content as a character stream.

Listing 3. Example of the ParsingReader

```
InputStream stream = ...; // the document to be parsed
Reader reader = new ParsingReader(parser, stream, ...);
try {
 ...; // read the document text using the reader
} finally {
 reader.close(); // the document stream is closed automatically
}
```

Document metadata

The final argument to the `parse` method is used to pass document metadata in and out of the parser. Document metadata is expressed as a metadata object.

Table 2 lists some of the more interesting metadata properties.

Table 2. Metadata properties

Property	Description
`Metadata.RESOURCE_NAME_KEY`	The name of the file or resource that contains the document — A client application can set this

	property to allow the parser to use file name heuristics to determine the format of the document. The parser implementation may set this property if the file format contains the canonical name of the file (the GZIP format has a slot for the file name, for example).
`Metadata.CONTENT_TYPE`	The declared content type of the document — A client application can set this property based on, such as an `HTTP Content-Type` header. The declared content type may help the parser to correctly interpret the document. The parser implementation sets this property to the content type according to which the document was parsed.
`Metadata.TITLE`	The title of the document — The parser implementation sets this property if the document format contains an explicit title field.
`Metadata.AUTHOR`	The name of the author of the document — The parser implementation sets this property if the document format contains an explicit author field.

Note that metadata handling is still being discussed by the Apache Tika development team, and it is likely that there will be some (backwards-incompatible) changes in metadata handling before Tika V1.0.

Parser implementations

Apache Tika comes with a number of parser classes for parsing various document formats, as shown in Table 3.

Table 3. Tika parser classes

Format	Description
Microsoft® Excel® (application/vnd.ms-excel)	Excel spreadsheet support is available in all versions of Tika and is based on the HSSF library from POI.
Microsoft Word® (application/msword)	Word document support is available in all versions of Tika and is based on the HWPF library from POI.
Microsoft PowerPoint® (application/vnd.ms-powerpoint)	PowerPoint presentation support is available in all versions of Tika and is based on the HSLF library from POI.
Microsoft Visio® (application/vnd.visio)	Visio diagram support was added in Tika V0.2 and is based on the HDGF library from POI.
Microsoft Outlook® (application/vnd.ms-outlook)	Outlook message support was added in Tika V0.2 and is based on the HSMF library from POI.
GZIP compression (application/x-gzip)	GZIP support was added in Tika V0.2 and is based on the `GZIPInputStream` class in the

	Java 5 class library.
bzip2 compression (application/x-bzip)	bzip2 support was added in Tika V0.2 and is based on bzip2 parsing code from Apache Ant, which was originally based on work by Keiron Liddle from Aftex Software.
MP3 audio (audio/mpeg)	The parsing of `ID3v1` tags from MP3 files was added in Tika V0.2. If found, the following metadata is extracted and set: • `TITLE Title` • SUBJECT `Subject`
MIDI audio (audio/midi)	Tika uses the MIDI support in `javax.audio.midi` to parse MIDI sequence files. Many karaoke file formats are based on MIDI and contain lyrics as embedded text tracks that Tika knows how to extract.
Wave audio (audio/basic)	Tika supports sampled wave audio (.wav files, etc.) using the `javax.audio.sampled` package. Only sampling metadata is extracted.
Extensible Markup Language (XML) (application/xml)	Tika uses the `javax.xml` classes to parse XML files.
HyperText Markup Language (HTML) (text/html)	Tika uses the CyberNeko library to parse HTML files.
Images (image/*)	Tika uses the `javax.imageio` classes to extract metadata from image files.
Java class files	The parsing of Java class files is based on the ASM library and work by Dave Brosius in JCR-1522.
Java Archive Files	The parsing of JAR files is performed using a combination of the ZIP and Java class file parsers.
OpenDocument (application/vnd.oasis.opendocument.*)	Tika uses the built-in ZIP and XML features in the Java language to parse the OpenDocument document types used most notably by OpenOffice V2.0 and higher. The older OpenOffice V1.0 formats are also supported, although they are currently not auto-detected as well as the newer formats.
Plain text (text/plain)	Tika uses the International Components for Unicode Java library (ICU4J) to parse plain text.
Portable Document Format (PDF) (application/pdf)	Tika uses the PDFBox library to parse PDF documents.
Rich Text Format (RTF) (application/rtf)	Tika uses Java's built-in Swing library to parse RTF documents.
TAR (application/x-tar)	Tika uses an adapted version of the TAR parsing

	code from Apache Ant to parse TAR files. The TAR code is based on work by Timothy Gerard Endres.
ZIP (application/zip)	Tika uses Java's built-in ZIP classes to parse ZIP files.

You can also extend Apache Tika with your own parsers, and any contributions to Tika are welcome. The goal of Tika is to reuse existing parser libraries like Apache PDFBox or Apache POI as much as possible, so most of the parser classes in Tika are adapters to such external libraries.

Apache Tika also contains some general-purpose parser implementations that are not targeted at any specific document formats. The most notable of these is the `AutoDetectParser` class that encapsulates all Tika functionality into a single parser that can handle any type of document. This parser will automatically determine the type of the incoming document based on various heuristics and will then parse the document accordingly.

Now it's time for hands-on activities. Here are the classes we will develop throughout our tutorial:

1. `BudgetScramble` — Shows how to use Apache Tika metadata to determine which document has been changed recently and when.
2. `TikaMetadata` — Shows how to get all Apache Tika metadata of a specific document, even if there is no data (just to display all metadata types).
3. `TikaMimeType` — Shows how to use Apache Tika's mimetypes to detect the mimetype of a particular document.
4. `TikaExtractText` — Shows Apache Tika's text-extraction capabilities and saves extracted text as an appropriate file.
5. `LanguageDetector` – Introduces the Nutch language's identification ability to identify the language of particular content.
6. `Summary` – Sums up Tika features, such as `MimeType`, content charset detection, and metadata. In addition, it introduces cpdetector functionality to determine a file's charset encoding. Finally, it shows Nutch's language identification in process.

Requirements

- Ant V1.7 or higher
- Java V1.6 SE or higher

Section 2. Lesson 1: Extracting metadata from a PDF file

So you've got Apache Tika downloaded and installed locally. The question now is, what do you do with it? We suggest taking advantage of Tika's parsing abilities to extract some metadata from your favorite PDF file. We randomly chose the FY2010 budget for the U.S. National Aeronautics and Space Administration (NASA).

Let's begin with some basic preparatory steps:

1. Build yourself a copy of tika-app. The easiest way to do this is to unpack your copy of apache-tika-X.Y-src.zip and change directory into the unpacked directory. From there, type `mvn` package.
2. Ensure that everything built correctly. Type `java –jar tika-app/target/tika-app-X.Y.jar –h`. If you see output similar to Listing 4, you are good to go.

Listing 4. Output from Java command

```
java –jar tika-app/target/tika-app-X.Y.jar –h

usage: tika [option] [file]

Options:
    -?  or --help        Print this usage message
    -v  or --verbose     Print debug level messages
    -g  or --gui         Start the Apache Tika GUI
    -eX or --encoding=X  Use output encoding X
    -x  or --xml         Output XHTML content (default)
    -h  or --html        Output HTML content
    -t  or --text        Output plain text content
    -m  or --metadata    Output only metadata

Description:
    Apache Tika will parse the file(s) specified on the command line and output the
    extracted text content or metadata to standard output.
    Instead of a file name you can also specify the URL of a document to be parsed.
    If no file name or URL is specified (or the special name "-" is used), then the
    standard input stream is parsed.
    Use the "--gui" (or "-g") option to start the Apache Tika GUI.
    You can drag and drop files from a normal file explorer to the GUI window to extract
    text content and metadata from the files.
```

Determining what metadata is available

Before delving too deeply into Apache Tika's rich Java API, the first step is to figure out what and how much metadata is available from the PDF file. *Metadata,* used to refer to "data about data," is a description of a particular content item (in this case, the PDF file), typically consisting of a set named fields, each of which contains metadata values. As an example, a PDF file may have a metadata description that includes an author field, with a value of Barack Obama. We can use the aforementioned command-line utility of Tika to determine what metadata is available from the PDF file, as in Listing 5.

Listing 5. Reading PDF metafile data

```
java –jar tika-app/target/tika-app-X.Y.jar –m \
    ./National_Aeronautics_and_Space_Administration.pdf

Content-Type: application/pdf
Last-Modified: Tue Feb 24 04:56:17 PST 2009
created: Sat Feb 21 07:38:41 PST 2009
creator: Adobe InDesign CS4 (6.0)
producer: Adobe PDF Library 9.0
resourceName: National_Aeronautics_and_Space_Administration.pdf
```

The above output gives us a preview of what metadata is available from the downloaded PDF file. Unfortunately, outside of the last modified time and created time, there isn't a lot of interesting metadata available. Looking at the PDFs available on the Whitehouse budget site (http://www.whitehouse.gov/omb/budget/Overview/, we asked ourselves, "Which budget was uploaded (or modified) last?" And was it because there was that much indecisiveness on it? Perhaps there were budget increases (or decreases) that needed to be factored in at the last minute. In any case, this type of question can easily be answered by whipping together a quick Tika-based Java program. (OK — the rationale behind the budget increases can't, but we digress.)

Lesson 1 helps you determine a document that has been changed or modified recently. It is important to remember that the documents are located remotely on the web.

Listing 6. determineLast.java

```
public void determineLast() throws Exception {
 Tika tika = new Tika();
 Date lastDate = new Date();
 lastDate.setYear(lastDate.getYear() - 10);
 String lastUrl = null;
 for (String budgetUrl : URLs) {
 Metadata met = new Metadata();
 try {
 tika.parse(new URL(budgetUrl).openStream(), met);
 Date docDate = BudgetScramble.toDate(met.get(LAST_MODIFIED));
```

```
 log.info(System.getProperty("line.separator") + "Metadata:\t" +met);
 if (docDate.after(lastDate)) {
 lastDate = docDate;
 lastUrl = budgetUrl;
 }
 } catch (Exception e) {
         log.error(e.getLocalizedMessage() + " " + e.getCause());
 }
    }
```

You can run this example by typing `ant budgetscramble`. Listing 7 shows the result of running this program.

Listing 7. Result of Ant command

```
budgetscramble:
     [java] 09/12/23 09:29:08 INFO example.BudgetScramble: The last budget to be
 finished is...[http://www.whitehouse.gov/omb/asset.aspx?AssetId=807] on: Thu Fe
b 26 15:55:07 IST 2009
```

Listing 8 shows another use of Apache Tika, in which we can find out exactly to what document this maps.

Listing 8. Tika mapping example

```
java -jar tika-app/target/tika-app-X.Y.jar \
    -x "http://www.whitehouse.gov/omb/budget/Overview/" | grep Asset | grep 807
<a href="http://www.whitehouse.gov/omb/asset.aspx?AssetId=807">Presentation and
Technical Changes</a>
```

Interestingly enough, a set of new changes and appropriations are in this fiscal year's budget.

The above example serves to illustrate the ease with which metadata can be extracted from content using Apache Tika. Of course, your mileage may vary and Tika strives to extract as much provided metadata as possible. Certainly, this does not limit the ability of Tika to extract derived metadata. The `org.apache.tika.metadata.Metadata class` provides a rich structure with the ability for merging and for easily adding new metadata keys, and for replacing and amending those that are already extracted. To date, Tika supports more than 20 common formats, including Microsoft Word and Excel, ZIP/GZIP, and others. It is best to check http://lucene.apache.org/tika/formats.html for the most up-to-date list (or the earlier part of this article where we expressed this information).

As Lesson 1 illustrates, Tika is a facade class for accessing Tika functionality. This class hides much of the underlying complexity of the lower-level Tika classes and provides simple methods for many common parsing and type-detection operations.

Metadata is a multi-valued metadata container. The most interesting method is a

parse that gets two parameters: `InputStream` and `metadata`.

Section 3. Lesson 2: Automatic metadata extraction from any file type

Despite the previous PDF files from Lesson 1, Apache Tika has the ability to arbitrarily extract metadata from any file type. You'll learn exactly how it does this in the coming lessons. If you can't wait, jump to lessons 3 and 4, but to illustrate this, we'll take an arbitrary Open Office Document Template (.odt file) and print some of its metadata to the console automatically. This process can be executed for any file or content type in general, regardless of whether Tika actually understands what type it is. Tika's goal is to extract as much minimal metadata information as possible from the underlying file type, as we'll see in Listing 9.

Listing 9. Extracting metadata with Tika

```
List<File> list =
Utils.getFiles(new File(Messages.getProperty("m001")), new ArrayList<File>());
for (File f : list) {
try {
        TikaMetadata tm = new TikaMetadata(f);
        tm.showMe();
 } catch (Exception e) {
        log.error(e.getLocalizedMessage() + " " + e.getCause());
                }
}
```

First, we get a list of files with which we're going to work. Second, for each file, we define the `TikaMetadata` object and show a metadata of the file.

Type the following and see what happens: `ant tikametadata`. The partial output appears in Listing 10.

Listing 10. ant listing of TikaMetaData

```
     [java] thai_odt.odt
     [java] nbObject=0 nbPara=5 nbImg=0 nbTab=0 generator=OpenOffice.org/3.1$Lin
ux OpenOffice.org_project/310m19$Build-9420 date=2009-12-13T08:29:31 nbWord=1516
 nbPage=1 Content-Type=application/vnd.oasis.opendocument.text creator=Oleg  nbC
haracter=2031
```

Note that the above presents a set of metadata keys (present before the = sign in the above output), with associated values (present after the = sign in the above output) associated with the file type. Since Tika has the ability to detect and parse

.odt files, it was able to extract more comprehensive metadata (e.g., generator, nbWord(s), nbPage(s), etc.)

Section 4. Lesson 3: Understanding mimetypes

So, how did Apache Tika figure out how to extract text and metadata from the PDF budget files in Lesson 1? Tika comes with a comprehensive mimetype repository. A mimetype repository is a set of definitions of the standard Internet Assigned Numbers Authority (IANA) mimetypes, where, for each mimetype defined, an entry is recorded containing:

- Its names (including aliases)
- Its parent and child mimetypes
- Mime MAGIC, a set of control bytes used to compare the first 1,024 KB of the file for detection
- URL patterns, matching the file extension or file name of a file being analyzed
- XML root characters and namespaces

Apache Tika uses the mimetype repository and a set of schemes (any combination of mime MAGIC, URL patterns, XML root characters, or file extensions) to determine if a particular file, URL, or piece of content matches one of its known types. If the content does match, Tika has detected its mimetype and can proceed to select the appropriate parser. In this lesson, we'll explore some of the properties of a mimetype for a particular file and print those properties out. Often when manipulating files, we need to know what the file mimetype (e.g., a TXT file, HTML, or PDF), and how to read it as-is, or like a binary. Simply binary output gives nothing. According to the mimetype, you could choose an appropriate parser or something that relates to your business.

Listing 11. Working with mimetypes

```
public static void main(String[] args) {
Metadata metadata = new Metadata();
MimeTypes mimeTypes = TikaConfig.getDefaultConfig().getMimeRepository();
List<File> list = Utils.getFiles(new File(Messages.getProperty("m001")),
new ArrayList<File>());
String mime = null;
for (File f : list) {
 URL url;
 try {
        url = new URL("file:" + f.getAbsolutePath());
        InputStream in = url.openStream();
```

```
 mime = mimeTypes.detect(in, metadata).toString();
 log.info("Mime: " + mime + " for file: " + f.getName());
 } catch (Exception e) {
         log.error(e.getLocalizedMessage() + " " + e.getCause());
 }//try-catch
 }//foreach
}//function
```

```
ant tikamimetype
```

Section 5. Lesson 4: Automatic text extraction from any file type

Besides having the ability to extract metadata, Apache Tika also strives to provide textual content, independent of other extraneous information (packaging, headers, binary garble, and other miscellaneous information typically packaged along with files) for any file type, so long as it can parse it. Tika's parsers all must implement a basic means for stripping out the text from a particular file type it is able to parse. Textual content is useful as it can be sent to search engines, indexed in content-management systems and used to show summaries of information for particular pieces of content. In the example below, we'll show in a few steps how easy it is in Tika to extract textual content from any file type. And, as opposed to the normal disclaimers seen on TV, we do want you to try this at home.

Listing 12. Extracting textual content from a file type

```
TikaConfig tc = TikaConfig.getDefaultConfig();
List<File> list = Utils.getFiles(new File(Messages.getProperty("m001")),
new ArrayList<File>());
Utils.deleteFiles(new File(Messages.getProperty("m002")));
for (File f : list) {
 try {
 String txt = ParseUtils.getStringContent(f, tc);
      Utils.writeTxtFile(new File(Messages.getProperty("m002")
+ File.separator + Utils.getFileNameNoExtension(f) + ".txt"),txt);
      log.info(Messages.getProperty("m003") + f.getName()
+ Messages.getProperty("m004"));
 } catch (TikaException e) {
        log.error(e.getLocalizedMessage() + " " + f.getName());
 } catch (IOException e) {
        log.error(e.getLocalizedMessage() + " " + f.getName());
 }catch (Exception e) {
        log.error(e.getLocalizedMessage() + " " + f.getName());
 }
}
```

The magic is done by the `ParseUtils getStringContent(...)` function.

`ParseUtils` contains utility methods for parsing documents. It is intended to provide simple entry points into the Tika framework. One of the parameters is a file, and the second is `TikaConfig`, which parses XML configuration files. It's simple and powerful.

Are you burning with curiosity to know how this magic trick is done? Type `ant tikaextracttext`.

Section 6. Lesson 5: Language identification

To have content is a good start, but it's not enough. The knowledge of language is missing. Did you ever think about how to identify content language? In general, the Natural Language Processing approach deals with these issues. Unfortunately, you have to be acquainted with that. But it's really not so bad. Nutch has developed a module called `LanguageIdentifier`, which we are going to use. Let's see how it works.

Listing 13. Example of using LanguageIdentifier

```
List<File> list = Utils.getFiles(new File(Messages.getProperty("m001")),
new ArrayList<File>());
LanguageDetector ld = null;
for (Iterator<File> iterator = list.iterator(); iterator.hasNext();) {
 File file = (File) iterator.next();
 ld = new LanguageDetector(file);
    log.info(Messages.getProperty("m072") + ld.getLanguage() +
Messages.getProperty("m073") + ld.getFile().getName());
}//for
```

See how easy it is? Just call the `getLanguage()` function and *voilà!*

Don't hesitate to run this example: `ant languagedetector`.

That's it. If you're interested to know how language identification works, how to add a new language profiler, and even more, read further.

All languages have been identified properly except Chinese. Why? Because our system doesn't have the capability to recognize a new language that's out of the box. Therefore, let's start to create an N-gram profiler. Before you jump into deep water, we would like to explain what an N-gram is, how it works, and how to build a training set.

What is an N-gram?

N-grams are sequences of characters or words extracted from text or documents. They could be divided into two groups: character-based or word-based. An N-gram is a set of *N* consecutive characters extracted from a word, or in our case, string. The motivation behind that is similar words will have a high proportion of N-grams. The most common values for *N* are 2 and 3, called bigrams and trigrams respectively. For instance, the word TIKA results in the generation of the bigrams *T, TI, IK, KA, A* and trigrams: **T, *TI, TIK, IKA, KA*, A**. The "*" denotes a padding space. Character-based N-grams are used in measuring the similarity of character strings. Some applications using character-based N-grams are spelling checker, stemming, and OCR.

As you can guess, word N-grams are sequences of *N* consecutive words extracted from text. It is also language-independent. The N-gram based similarity between two strings is measured by Dice's coefficient (informally, similarity measure). s = (2|X /\ Y|)/(|X| + |Y|), where *X* and *Y* are the sets of characters. The /\ means an intersection between two sets. If we take as a string-similarity measure, the coefficient may be calculated for two strings or words, *x* and *y* using bigrams: s = (2Nt)/(Nx + Ny), where *Nt* is the number of character bigrams found in both strings, *Nx* is the number of bigrams in string *x* and *Ny* is the number of bigrams in string *y*. For example, to calculate the similarity between TIKA and TECA, we would find the set of bigrams in each word as follows: {TI, IK, KA} and {TE, EC, CA}. Each set has three elements, and the intersection of these two sets has only zero. Now putting this into formula and calculate s = (2x0)/(3+3) = 0. Is totally dissimilar for bigrams. You'll get other results for 1-gram.

A large text corpus (training corpus) is used to estimate N-gram probabilities. In Nutch's language identification, the file comes with an N-Gram Profile (NGP) extension. It's a file that contains N-grams and its scores. For instance, _dé17376 is a trigram with score 17376.

One of the major problems of N-gram modeling is its size. Fortunately, we only have to fulfill the process once. Another interesting example of using N-gram is the extracting features for clustering large sets of satellite Earth images and for determining what part of the Earth a particular image came from.

How is it possible to identify language?

Generally speaking, when a new document comes whose language is to be identified, we first create an N-gram profile of the document and calculate the distance between the new document profile and the language profiles. The distance is calculated according to "out-of-place measure" between the two profiles. The shortest distance is chosen, and it is predicted that the particular document belongs to that language. A threshold value has been introduced so that if any distance goes above the threshold, the system claims that the language of the document cannot be determined or mistakenly determined. Before we created zh.ngp, our system determined Chinese documents as German.

By adding a new N-gram language profile, we can get the language identification correctly. Apache Tika V0.5 has a `LanguageIdentifier` module plugged-in framework. It works fine unless a document does not contain text `LanguageIdentifier` couldn't recognize as one of its supported languages. So we've separated it to different packages. Now you can manipulate as you wish, add any language that is still unsupported and use a call to the `identify(...)` function from your code.

One of the parameters the `NgramProfiler` main function expects to obtain is a TXT file. In our case, it should be a text file containing Chinese text taken from Wikipedia. The amount of text ought to be large in order to create an N-gram profile that could predict with high probability what language that file's content belongs to. In addition, text is needed to be taken from the various wiki topics. The topic might be geography, mathematics, astronautics, etc. It is also important to reduce the noise (such as exclude links, image names, and special characters). The data have to be redundant, preventing overlapping in consideration of identification accuracy.

Create a TXT file, such as chines4ngram.txt. Go to Wikipedia, then copy and paste the text into chines4ngr.txt. Try to avoid leaving blank lines. Crawl through the links and gather stuff. More is better in this case. It's a boring and exhausting process, but it's important; 5,000-6,000 lines of text will be enough.

Note: This process could be automated by using Nutch crawler.

`NgramProfiler`'s main function expects to get parameters as follows: `-create <name_of_gram_profile> <text_file>`.

Using Ant, type:

```
ant createngram -Dngpname=/home/olegt/tutorial/zh \
-Dfile="/home/olegt/ chines4ngram.txt -Dencoding=utf-8
```

After a while, copy zh.ngp to the org.apache.analysis.lang package and rerun `TikaLanguageIdentifier` by typing `ant TikaLanguageIdentifier`. Look at the output. All content from Chinese files has been identified as zh (Chinese).

In this tutorial, we have used an additional framework called cpdetector to determine a file's charset encoding. The name cpdetector is a short form for code page detector and has nothing to do with Java classpaths. The cpdetector is a framework for configurable code page detection of documents. It may be used to detect the code page of documents retrieved from remote hosts. Code page detection is needed whenever it is not known which encoding a document belongs to. Therefore, it is a core requirement for any application in the field of information mining or just information retrieval.

Resources

Learn

- Visit Apache.org/tika to learn more.
- Learn more about Nutch.
- Be sure to check out cpdetector.
- To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.
- Stay current with developerWorks' Technical events and webcasts.
- Follow developerWorks on Twitter.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products, as well as our most popular articles and tutorials.
- The My developerWorks community is an example of a successful general community that covers a wide variety of topics.
- Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

- Innovate your next open source development project with IBM trial software, available for download or on DVD.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the authors

Oleg Tikhonov

Originally from Kazakhstan, Oleg graduated from Ariel University Center of Samaria in Israel. He has worked in the software industry for more than 5 years. At IBM, Oleg develops automated software test solutions. In his spare time, when not exploring the details of new J2EE technologies and how to stretch their capabilities, he likes diving with friends and spending time with his beautiful wife, Vera.

Chris Mattmann

Chris Mattmann is a senior computer scientist at NASA's Jet Propulsion Laboratory, working on instrument and science data systems on Earth science missions and informatics tasks. Mattmann has a joint appointment as an adjunct assistant professor in the University of Southern California (USC) Computer Science Department, where he teaches a course on software architecture to local and remote graduate students over USC's distance education network. His research interests are primarily software architecture and large-scale data-intensive systems. He received his Ph.D. in computer science from USC. He is a member of the IEEE.