Build a text visualization and analytics application
Develop a word-wave application with Eclipse and other open source software

Sami Salkosuo
April 02, 2013
(First published March 19, 2013)

Text visualization is an effective way to see and analyze what a designated text is saying. Learn to combine Eclipse and open source text visualization and analytics tools to build a word-wave application that visualizes and compares two texts.

02 Apr 2013 - Updated URLs for IBM InfoSphere Streams and IBM InfoSphere BigInsights links in Get more from text and Resources > Learn.

Get more from text

Do even more with text by extracting key information and concepts. Advanced text analytics capability in the IBM big data platform includes a toolkit with accelerators, an IDE, and a declarative language (AQL) that enable developers to parse text, find the elements that are search targets, understand their meaning, and extract them in a structured form for use in other applications. The advanced text analytics capability is available as part of the IBM InfoSphere Streams and IBM InfoSphere BigInsights products. Learn more about InfoSphere Streams and InfoSphere BigInsights.

Text visualization is a powerful and sometimes eye-opening way to ascertain quickly what a specific text is saying. As a by-product, visualization also provides a means for ad hoc analysis of a text or texts. The article shows how to develop text visualization and analysis software with open source tools and libraries. The application compares and analyzes two texts with the same or a similar context, enabling users to gain new insight into those texts or their context.

The application that you build is based on word cloud visualization. A word cloud visualization analyzes a specific text and ranks its words in proportion to how frequently they occur. The ranked words are then sized based on their rankings; the highest-ranking word display in the largest font in the visualization. Placement of words in the visualization can vary, but typically they resemble a cloud, as in Figure 1:
Figure 1. A word cloud

To generate the word cloud in Figure 1, I used IBM Many Eyes (see Resources) to analyze the letter from the President and CEO in the IBM 2011 Annual Report.

The application in this article generates a word wave—a text visualization that is shaped like a wave. That word wave places high-ranking words in the upper left corner. Figure 2 shows an example that uses the same text that is visualized in Figure 1:

Figure 2. Sample word wave

Visualizing a text reveals high-ranking words. Analysis of the text that is based on the visualization assumes that high-ranking words have a hierarchy of importance. Comparison comes into play when two text visualizations are displayed together. If the contexts of both texts are the same or similar, the comparison is especially meaningful. For example, a comparison of texts that describe the strategies of two companies in the same industry would reveal similarities and differences between those two companies' priorities.

Figure 3 is a rough sketch of a final comparison of two texts. The visualization of the first text is on top, and the visualization of the second text is on the bottom. The high-ranking words in both are on the left side.
The goals of the article and its code are to show you how to:

- Develop a command-line application for visualizing and comparing texts with open source tools and libraries.
- Create a visualization (similar to Figure 2) of a designated text with a word-wave visualization.
- Combine two visualizations in the same image for comparison and analysis.
- Create a visually appealing video from the visualization.

The article does not delve deeply into development details, so experience with Java™ development and the Eclipse programming model is helpful for readers. All of the application source code — the Eclipse projects for the application and a ready-to-deploy update site — are available for download.

I start with an overview of the components of the development environment.

### Prerequisites

To complete the steps in this article, you must install Eclipse and download all the necessary libraries. I used the following versions:

- **Eclipse Juno**, v4.2
- **Command Line Program**, v1.2.0
- **Processing**, v1.5.1
- **WordCram**, v0.5.6
- **Monte Media Library**, v0.7.7

### Development environment

The development environment consists of various open source tools and libraries that, in combination, make it easy to create word waves and videos. The application code itself is relatively short. The tools and libraries handle the heavyweight processing of images, videos, and the command-line interface.
Eclipse

When you use Eclipse as an IDE, you can take advantage of Command Line Program.

Command Line Program

Command Line Program is a third-party Eclipse Rich Client Platform (RCP) application for creating command-line applications. It uses the Eclipse plug-in programming model, including features and update sites (see Resources for a link to Eclipse concepts). Command Line Program provides the infrastructure for command-line applications, such as command-line parsing, a help command, logging, and other basic functions. The plan is for you to develop visualization and analysis software as an extension command to Command Line Program.

Processing and WordCram

Processing is both a language and development environment for creating images and animations. For an introduction to Processing, see "Data visualization with Processing, Part 1." Processing is easy to learn and use, yet you can accomplish great things with it (see Resources for a link to examples at OpenProcessing.org). WordCram is a customizable library for Processing for creating word clouds.

Monte Media Library

Monte Media Library is a simple, excellent open source library for writing, reading, and manipulating video and images. The author, Werner Randelshofer, says on his website that the Monte Media Library is experimental for his personal studies. Fortunately, though, he decided to publish it so others can use it too. Unlike other available libraries that seem cumbersome and require native code, Monte Media Library is easy to use and is pure Java code.

Developing the application

Now that you are acquainted with the tools, you can start to develop an application named TextVisualizationAndAnalysis and add an extension command named compare— to Command Line Program.

You need the Command Line Program project so you can develop the extension to it. Download the CLP_Plugin_Main project and import it into the Eclipse workspace. The project includes the source code for the Command Line Program, and provides extension points for you to develop extensions in your own projects.

Create a plug-in project

To develop an extension command to Command Line Program, first create a plug-in project, as you would with any other Eclipse extension project:

1. Create a new plug-in project and enter TextVisualizationAndAnalysis as the project name.
2. Select 3.5 or greater as the Eclipse version and click Next.
3. Enter 0.0.1 as the version number, clear (if selected) **This plug-in will make contributions to the UI**, and accept defaults in other fields. Click **Next** to get to the final screen.

4. Clear (if selected) **Create a plug-in using one of the templates**, and click **Finish** to create the project.

5. Your new project is now in the Eclipse workspace, and the plugin.xml file is open in the screen. (If not, and if plugin.xml does not exist in the project directory, open the META-INF/MANIFEST.MF file instead).

6. Open the Dependencies tab and in the Required Plug-ins section add `com.softabar.clpp.application` to the project. This plug-in comes with the Command Line Program Eclipse project that you imported into the Eclipse workspace.

7. Go to the Extensions tab and add an extension that extends the `com.softabar.clpp.application.command` extension point, which is provided by Command Line Program.

8. Enter the extension details. Enter `compare` in the **name** field. Enter `Visualize and compare two text files` in the **help** field. Enter `textvisualizationandanalysis.CompareCommand` in the **class** field. See Figure 4:

**Figure 4. New command for Command Line Program**

Create the class later. Also, add new information to the plug-in later, as needed.

**Add required libraries**

To create the **compare** command, add these libraries to the plug-in:

- `core.jar`, from Processing
- `WordCram.jar`, `jsoup-1.3.3.jar`, and (optionally) `cue.language.jar` from WordCram
- `monte-cc.jar`, from the Monte Media Library

To add them:

1. Create a lib directory in the plug-in project and add the JAR files to that directory.
2. To add the lib directory to the plug-in build, open plugin.xml and select the **Build** tab. To select the lib directory in the Binary Build dialog, click its check box, as in Figure 5:
3. Open the Runtime tab, and in the Classpath dialog, select **Add** to add the libraries to the plug-in classpath. Figure 6 shows the Classpath dialog with all the libraries (not including cue.language.jar) added:

**Figure 6. Adding the libraries to the plug-in classpath**

4. Save the plugin.xml file.

**Write the code**

Now you can write the actual code for the command. The source code for the command is in the listings that follow (with package and import statements intentionally omitted). You also must add some information to the plugin.xml file, which I show you after you go through the code listings.

Listing 1 contains the class declaration and variables:

**Listing 1. Class declaration and variables**

```java
public class CompareCommand extends PApplet implements ICommand, WordColorer {
    private static final long serialVersionUID = -188083470351748783L;
    private static CLPPLogger logger = CLPPLogger.getLogger(CompareCommand.class);
    private static boolean testing = true;
    private static boolean processingDone = false;
```
In Listing 1, you extend the `processing.core.PApplet` class to take advantage of Processing methods. Then, you implement two interfaces: `com.softabar.clpp.program.ICommand` and `wordcram.WordColorer`. The `com.softabar.clpp.program.ICommand` interface is for Command Line Program; it is called by the Command Line Program when the command runs. The `wordcram.WordColorer` interface handles the colors of the word clouds (or waves). Some of the variables are declared `static` because they must be visible to Processing code during execution.

Listing 2 shows the `execute()` method from the `ICommand` interface:

```
Listing 2. The `execute()` method

public void execute(CommandLine commandLine, IProgramContext programContext) {
    testing = false;
    String inputFileStr = commandLine.getOptionValue("input1");
    inputTextFile1 = new File(inputFileStr);
    if (!inputTextFile1.exists()) {
        Output.error(inputFileStr + " does not exist.");
        return;
    }
    inputFileStr = commandLine.getOptionValue("input2");
    inputTextFile2 = new File(inputFileStr);
    if (!inputTextFile2.exists()) {
        Output.error(inputFileStr + " does not exist.");
        return;
    }
    drawTitle = commandLine.hasOption("title");
    fileName = commandLine.getOptionValue("filename", "results");
    outputDir = commandLine.getOptionValue("outputdir", ".");
    if (!outputDir.endsWith("/")) {
        outputDir = outputDir + "/";
    }
    title1 = commandLine.getOptionValue("title1", inputTextFile1.getName());
    title2 = commandLine.getOptionValue("title2", inputTextFile2.getName());
    String framerate = commandLine.getOptionValue("framerate", "5");
    frameRate = Integer.parseInt(framerate);
    createVideo = commandLine.hasOption("video");
    Output.println("Generating comparison word waves...");
    generateWordCloud();
    createVideo();
}
```
In Listing 2, the execute() method receives an org.apache.commons.cli.CommandLine instance as a parameter; the parameter gets the options for the command. Add the supported options to plugin.xml later. After you get and set options, create the word cloud by calling the generateWordCloud() method. Then, you create the video by calling the createVideo() method.

Listing 3 shows the generateWordCloud() method, which calls the main method in the processing.PApplet class and then waits until Processing/WordCram finishes rendering the word cloud:

**Listing 3. The generateWordCloud() method**

```java
private void generateWordCloud() {
    try {
        main(new String[]{"--present", getClass().getName()});
        // wait until word wave is finished
        while (!processingDone) {
            try {
                Thread.sleep(0, 1);
            } catch (InterruptedException e) {
            }
        }
    } catch (Exception e) {
        logger.error(e.toString(), e);
        Output.error(e.toString());
    }
}
```

Listing 4 shows the setup for generating word clouds:

**Listing 4. The setup() method**

```java
public void setup() {
    if (testing) {
        logger.debug("testing");
        inputTextFile1 = new File("c:/CocaCola_MissionVisionValues.txt");
        inputTextFile2 = new File("c:/PepsiCo_MissionVisionValues.txt");
        outputDir = "c:/output/";
        fileName = "results";
        drawTitle = true;
        createVideo = false;
        title1 = "Coke"; // inputTextFile1.getName();
        title2 = "Pepsi"; // inputTextFile2.getName();
    }
    logger.debug("frameWidth: {}, frameHeight: {}", frameWidth, frameHeight);
    size(frameWidth, frameHeight);
    logger.debug("setup");
    // create buffer to draw the upper word wave
    buffer1 = createGraphics(frameWidth, frameHeight / 2, JAVA2D);
    buffer1.beginDraw();
    buffer1.background(255);
    wordCram1 = initWordCram(inputTextFile1, buffer1);
    // create buffer to draw the lower word wave
    buffer2 = createGraphics(frameWidth, frameHeight / 2, JAVA2D);
    buffer2.beginDraw();
    buffer2.background(255);
    wordCram2 = initWordCram(inputTextFile2, buffer2);
    // set up font for titles
    fill(0);
    textFont(createFont(font, 40));
    textAlign(CENTER);
}```
In Listing 4, the `setup()` method is called by Processing before it starts drawing. Here you initialize the screen size and background color, and create graphic buffers where you draw the word cloud that is generated by WordCram. This code also specifies variables for testing from within Eclipse.

WordCram, the library responsible for generating word clouds, is initialized in Listing 5. You can specify aspects of the word cloud such as placement and colors. WordCram provides a few placers, such as the wave used here, plus a few colorers for words. Here you use your own colorer.

**Listing 5. The `initWordCram()` method**

```java
private WordCram initWordCram(File inputFile, PGraphics buffer) {
    WordCram wordCram = new WordCram(this);
    if (buffer != null) {
        wordCram = wordCram.withCustomCanvas(buffer);
    }
    // initialize WordCram with specified placer, text file, // colorer, and other details
    wordCram = wordCram.fromTextFile(inputFile.getPath());
    wordCram = wordCram.withColorer(this);
    wordCram = wordCram.withWordPadding(2);
    wordCram = wordCram.withPlacer(Placers.wave());
    wordCram = wordCram.withAngler(Anglers.randomBetween(-0.15f, 0.15f));
    wordCram = wordCram.withFont(createFont(font, 40));
    wordCram = wordCram.sizedByWeight(7, 52);
    wordCram = wordCram.maxNumberOfWordsToDraw(maxWords);
    return wordCram;
}
```

Listing 6 shows the word cloud is generated by the `draw()` method:

**Listing 6. The `draw()` method**

```java
public void draw() {
    logger.debug("Draw..");
    // draw one word at a time
    if (wordCram1.hasMore()) {
        // draw next word in upper word wave
        wordCram1.drawNext();
        buffer1.endDraw();
        image(buffer1, 0, 0);
        buffer1.beginDraw();
        // draw next word in lower word wave
        wordCram2.drawNext();
        buffer2.endDraw();
        image(buffer2, 0, frameHeight / 2);  
        buffer2.beginDraw();
    } else {
        buffer1.endDraw();
        buffer2.endDraw();
        image(buffer1, 0, 0);
        image(buffer2, 0, frameHeight / 2);
        listSkippedWords(inputTextFile1.getName(), wordCram1);
        listSkippedWords(inputTextFile2.getName(), wordCram2);
        noloop();
    // if no video then
    // save only last frame result
    if (!createVideo) {
        saveFrame(outputDir + fileName + ".png");
    }
```

```
Generation of the word cloud happens one word at a time. **Listing 6** uses two different buffers for two word clouds, and both buffers are then drawn on the screen. After the word cloud is complete, you end drawing and list any skipped words. If you create a video, you save each frame as an image; those images are used to generate the video.

The purpose of the `listSkippedWords()` method, in Listing 7, is to print a list of the words that cannot be placed in the visualization:

**Listing 7. The listSkippedWords() method**

```java
private void listSkippedWords(String desc, WordCram wordcram) {
    Word[] words = wordcram.getWords();
    int skipped = 0;
    // for each word check whether it was skipped
    List<String> skippedWords = new Vector<String>();
    for (Word word : words) {
        if (word.wasSkipped()) {
            int skippedBecause = word.wasSkippedBecause();
            if (skippedBecause == WordCram.NO_SPACE) {
                // increase number of skipped words
                // only if no space for word
                skippedWords.add(word.word);
                skipped++;
            }
        }
    }
    // print number of skipped words
    if (skipped > 0) {
        logger.debug("skippedWords: {}, {}", desc, skippedWords);
        Output.println(desc + ": no space for " + skipped + " words: "
                        + skippedWords);
    }
}
```

If any skipped words are returned, it potentially means that the visualization is missing important words. Later analysis that is based on the visualization might be misleading or even false. If any skipped words are returned, you can run the program again so it can try to place all words in the word cloud.

The `colorFor()` method in Listing 8 implements the `WordColorer` interface from WordCram. The method returns randomly chosen colors from a predefined list.
Listing 8. The `colorFor()` method

```java
public int colorFor(Word w) {
    int index = (int) random(colors.length);
    int colorHex = colors[index];
    int r = colorHex >> 16;
    int g = (colorHex >> 8) & 0x0000ff;
    int b = colorHex & 0x0000ff;
    logger.debug("R: {}, G: {}, B: {}", new Integer[] { r, g, b });
    return color(r, g, b);
}
```

Listing 9 shows `createVideo()`, the final method that is called in Listing 6:

Listing 9. The `createVideo()` method

```java
private void createVideo() {
    if (createVideo) {
        Output.println("Generating video...");
        try {
            File aviFile = new File(outputDir, fileName + ".avi");
            // format specifies the type of video we are creating
            // video encoding, frame rate, and size is specified here
            Format format = new Format(org.monte.media.FormatKeys.EncodingKey,
                                          org.monte.media.VideoFormatKeys.ENCODING_AVI_PNG,
                                          org.monte.media.VideoFormatKeys.DepthKey, 24,
                                          org.monte.media.VideoFormatKeys.MediaTypeKey, MediaType.VIDEO,
                                          org.monte.media.VideoFormatKeys.FrameRateKey,
                                          new Rational(frameRate, 1),
                                          org.monte.media.VideoFormatKeys.WidthKey, width,
                                          org.monte.media.VideoFormatKeys.HeightKey, height);
            logger.debug("Framerate: {}", frameRate);
            AVIWriter out = null;
            try {
                // create new AVI writer with previously specified format
                out = new AVIWriter(aviFile);
                out.addTrack(format);
                int i = 1;
                // read the first image file
                String frameFileName = String.format(fileName + "-%04d.png", i);
                File frameFile = new File(outputDir, frameFileName);
                while (frameFile.exists()) {
                    logger.debug("Frame filename: ", frameFileName);
                    // while frame images exist
                    // create a Buffer and write it to AVI writer
                    BufferedImage buf = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
                    ImageIO.write(buf, "png", frameFile);
                    out.write(0, buf);
                    // read next frame image
                    i++;
                    frameFileName = String.format(fileName + "-%04d.png", i);
                    frameFile = new File(outputDir, frameFileName);
                }
            } finally {
                if (out != null) {
                    out.close();
                }
            }
            Output.println("Done.");
        } catch (IOException e) {
        }
    }
}
```
Listing 9 demonstrates the simplicity of creating a video from images. You need only to specify a video format, and then create the video one image at a time.

Add command-line options to plugin.xml
The code is done now. Before you can deploy it, add the command-line options to the plugin.xml file so Command Line Program can parse them:

1. Open plugin.xml and open the Extensions tab.
2. Select compare-command, right-click, and select New > option.
3. The input1, input2, and video options are required for the application. input1 and input2 specify the filenames of the two text files you want to compare. video generates the video (or, if omitted, does not generate the video). Figure 7 shows the input1 option definition (Text 1 file to compare) and other options that you can add if wanted:

Figure 7. Command options

The Compare command for the Command Line Program is now finished, and you can deploy it.

Deploying the application
To use the TextVisualizationAndAnalysis application, you must deploy it to Command Line Program. Create a feature and an update site for the TextVisualizationAndAnalysis plug-in and then install it in Command Line Program.

Create a feature
Features in Eclipse are installable and updateable collections of plug-ins. Command Line Program uses standard Eclipse feature functionality to enable installation, updating, and uninstallation of new extension commands. The quick steps to create a feature for the compare command are:
1. Create a feature project and name it `TextVisualizationAndAnalysisFeature`.
2. Set the version number to 0.0.1 and click **Next**.
3. Select the `TextVisualizationAndAnalysis` plug-in and click **Finish**.

**Generate an update site**

Update sites include features that you can install into applications. An update site can be a local directory or a remote web server. To generate an update site for the TextVisualizationAndAnalysis program:

1. Create an update site project and name it `TextVisualizationAndAnalysisUpdateSite`.
2. In the Update Site Map page, under **Managing the Site**, select **Add Feature** to add `TextVisualizationAndAnalysisFeature`, as in Figure 8:
   
   **Figure 8. Update Site Map page**

3. Click **Build All**.

**Install the command**

You cannot run the command by itself because you must install it in Command Line Program. Installation is done with Command Line Program itself. Enter the `admin` command (which here assumes that the update site created is in the c:/workspace/ directory):

```
clp.cmd admin --install --dir='c:\workspace\TextVisualizationAndAnalysisUpdateSite'
```

**Run the application**

Here are a few sample commands that show how to run the application.

Generate and show a text-comparison image:

```
clp.cmd compare --input1='c:/path/file1.txt' --input2='c:/path/file2.txt'
```

Use the input filenames as a title for the visualization:
Build a text visualization and analytics application

### Results and analysis

**Video encoding**

Note in Listing 6 that PNG format is specified for encoding for the video. The video might not be viewable in all players; VLC, an open source multimedia player, can play it (see Resources).

Now you can put the TextVisualizationAndAnalysis application to work. I use it to visualize two similar texts: the publicly available mission statements of two corporations in the same industry — Coca-Cola Company and PepsiCo (see Resources). Then, I use the visualization to make an analysis. The assumption is that word waves will visualize what these corporations think is most important to them, now and in the future. (To get the source texts plus the full application code in the plug-in project, see Downloads.)

### Visualization

I used the last command in the preceding section, Run the application, to generate the image and video. Figure 9 is a visualization comparison of the two texts:

**Figure 9. Visualization of two similar texts**

The top word wave in Figure 9 visualizes the Mission, Vision & Values statement of Coca-Cola; the bottom one visualizes the Values & Philosophy statement of PepsiCo. The word waves show the...
50 most-used words in the texts, with the most frequently mentioned words in the upper-left corner. The font size decreases to the right, indicating that words in the lower right corner are mentioned less frequently than the words in the upper left.

In the sidebar, view a 10-second video of the word-by-word generation of the visualization

**Analysis**

To analyze the visualization comparison in Figure 9, assign the most importance to the words in the left-most quarter. I first draw a vertical line in the visualization about a quarter of the way in from the left side of each word wave, as in Figure 10:

**Figure 10. Analysis of two texts**

I assume that words on the left side of the line are what both corporations consider to be most important.

Coca-Cola seems to focus on its vision, and to have a roadmap to achieve that vision. It also seems to view the world as its business, and it values actions and the quality of its work to achieve its vision.

PepsiCo seems to say that its business purpose is company growth, including financial performance. It also seems to value the environmental and social aspects.

This quick comparison of the visualizations, then, shows how two corporations in the same industry can differ in their thinking. A reasonable conclusion from the analysis is that the Coca-Cola Company is more concerned about its place in the future world, and that PepsiCo is more concerned with the business and its growth.

**Updating the application**

At some point you might want to update the TextVisualizationAndAnalysis application. Updating uses the same mechanism as installation. Follow these steps to update the application:
1. Make the necessary changes to the code or other files.
2. Increment the version of the plug-in and the feature.
   - **Important**: Both the plug-in and the feature need a version change. Otherwise, the update mechanism fails to detect that changes were made.
   - The version number must increment at least in the service segment. For example, old version is 0.0.1 and new version is 0.0.2.
3. Add the new version to the update site and the build update site.
4. Run this command:
   ```
clp.cmd admin --update --dir='c:\workspace\TextVisualizationAndAnalysisUpdateSite'
```
5. Run the application normally.

**Conclusion**

Visualization is a powerful tool for gleaning new insights from texts. You used existing open source tools to develop a visualization application that compares and analyzes any two texts. The TextVisualizationAndAnalysis application is ready to compare any other types of texts — for example, corporate strategies, biographies of celebrities, or works of fiction. Better yet, you can use the tools and techniques you learned about here to create your own visualization applications.
# Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug-in projects, ready to import into Eclipse</td>
<td>TextVisualizationAndAnalysis_projects.zip</td>
<td>1.3MB</td>
</tr>
<tr>
<td>Update site for deploying to Command Line Program</td>
<td>TextVisualizationAndAnalysisUpdateSite.zip</td>
<td>1.3MB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2013  
Trademarks  
(www.ibm.com/developerworks/ibm/trademarks/)