Chirographum Design Document

Chirographum Project

October 21, 2008

Contents

Chapter 1

Overview

ChiroGraphum is a framework and suite of tools for writer-dependent handwriting recognition. It will provide facilities for the developers to experiment with algorithms, as well as an interface for the user to apply combinations of algorithms to recognise their handwriting. It will use a pipeline architecture, and collections of feature detectors and classification algorithms.

Chapter 2

Pipeline

2.1 General Description

A Pipeline is a list of modules. The pipeline will be given data which will be passed through associated pipeline modules. The result of the final pipeline modules execution will be passed out of the pipeline as the pipelines overall result.

Within the pipeline, the result of one modules execution will be passed as the input data for the next module.

2.2 Pipeline Modules

A pipeline module is a unit of work done in a pipeline. All pipeline modules must implement the IPipelineModule interface.

A Pipeline Module will be passed input data in the form of an Object. It is the modules responsibility to ensure that the Object can be converted to what it requires.

A module may also be passed a second Object to be used as its configuration settings. Like with input data, it is the responsibility of the module to make sense of this object.

A Pipeline should be able to be paused and continued as appropriate. This will be provided by sleeping the current thread.

2.3 Pipeline Module Interface (IPipelineModule)

The IPipelineModule interface is to be implemented by all classes will be used as a module in a pipeline.

```
package comp314.interfaces;

/**
 *
 * @author Daniel Buchanan
```

```
 *
 */
public interface IPipelineModule {
    /**
     * Sets the module's input data.
     * @param data Input data
     */
    void setData(IData data);

    /**
     * Runs the module and returns its result.
     * @return result
     */
    IData run();

    /**
     * Set input data and run module.
     *
     * @param data Input data
     * @return Output data
     */
    IData run(IData data);

    /**
     * Sets the status reporter.
     * @param reporter Status Reporter
     */
    void setReporter(IReporter reporter);

    /**
     * Gets the input data.
     * @return the input data
     */
    IData getData();

    /**
     * Gets the result, after you have called run().
     * @return the result
     */
    IData getResult();

    /**
     * Sets module options.
     * @param options Module Options
     */
    void setOptions(IData options);
}
```

2.3.1 void setData(Object)

This method sets the input data for the pipeline module. It takes a single object to be used as data. The module will convert this to what ever data type it is designed to operate on. Should it be given incorrect data type it will thrown an exception.

Every time this method is called it will change the previously set data object to the one specified.

2.3.2 Object run()

This method causes the module to perform its task and then return the result as an Object.

2.3.3 Object run(Object)

Overloaded method which combines the functionality of the run() method and the setData(Object) method.

2.3.4 void setReporter(IReporter)

Sets a class which implements a status reporting function. This is used by the module to report status information (task progress, etc).

If the reporter is null it will not report anything.

2.3.5 Object getData()

Returns the data object set with either run(Object) or setData(Object) methods.

2.3.6 Object getResult()

Returns the result of running either run() or run(Object). It can only be called after one of the two methods has completed execution. If called before then it will return null. If called after execution it will return the same value as returned by run() or run(Object)

2.3.7 void setOptions(Object)

Sets options for the modules options. The object parameter is to be interpreted by the module - its content is not specified by this specification.

2.4 Status Reporter Interface (IReporter)

The IReporter interface is used to report status and progress information about a modules execution.

```
package comp314.interfaces;

/**
 *
```

```
 * @author Daniel Buchanan (replace with whoever actually created this)
 *
 */
public interface IReporter {
    /** Set status message.
     *
     * @param status Status message
     */
    void setStatus(String status);

    /** Sets progress level.
     *
     * @param progress Progress level
     */
    void setProgress(int progress);
}
```

2.4.1 void setStatus(String)

This method sets a status message. Possible uses include logging.

2.4.2 void setProgress(int)

This method is used to set the progress of the modules execution. Possible uses include driving a progress bar.

Chapter 3

Preprocessing

The softwares input image will be passed through a series of (some optional) preprocessing modules. The pipeline will be started off with the input images filename. This will be passed through image loading modules before being passed through any optional extra preprocessing modules.

3.1 Preprocessing Modules

3.1.1 Image Loading

This is a required module which will be at the start of the pre-processing pipeline. It takes a filename string as input and passes out the loaded image file in bitmap format.

Reading File

Possible options for this include:

1. Reading the file entirely into memory at the beginning before any operations are attempted.
2. Reading the file into memory as required for each read, i.e. memory mapped IO.

The option that will be used however depends on how the higher level image classes work, they may require the image to be loaded into memory entirely befire processing can occur, if this is the case the amount of memory used could become an issue.

Format Conversion

The file once loaded into memory should be converted into a format which the program is able to manipulate or work with easily, this could be for example a simple black and white bitmap.

The image file should be in one of the following formats:

1. JPEG format (jpg).

2. Graphics Interchange Format (gif).
3. Tagged Image File Format (tiff).
4. Portable Network Graphics (png).

The file format converter should be able to detect what format the input image is in, and covert it to an intermediary format such as BMP and the perform the conversion to whatever the internal image format is.

This should all be accomplished without modification of the original file

3.1.2 Black and White Conversion

This module will take in a color image (in memory) and convert it into a black and white binary image, which is able to be used internally.

It also performs noise reduction on the image by checking adjacent pixels to the current pixel and taking an average of those pixels.

Chapter 4

Image segmentation

In order to easily provide the user with the option to review the document layout automatically detected by the software, image segmentation is done in multiple stages. These consist of layout detection, user verification and splicing.

4.1 Data Types

4.1.1 ImageSegment

This data type stores information about an image segment. These fields include:

1. Co-ordinate for the top left corner of the segment (required)
2. Co-ordinate for the bottom right corner of the segment (required)
3. The bitmap image segment taken from the co-ordinates (optional)

Instances of the ImageSegment class will normally be produced by the Layout Detection stage of image segmentation. The bitmap field will be left as null until the Image Splicing segmentation stage.

4.1.2 SegmentLayout

This data type stores ImageSegments in the order they appear in the Source Image. It divides ImageSegments into lines in the order they appear in.

4.2 Layout Detection

This stages purpose is to detect each word in the input image. Its input consists of only the Source Image and it outputs a SegmentLayout. The SegmentLayout is an ordered list of lines detected in the Source Image. Each line is an ordered list of the detected words, each represented as an ImageSegment.

The ImageSegment instances produced by this stage will contain nothing more than the two co-ordinates which make up the box around the represented image segment. These co-ordinates may be adjusted in the User-verification stage before being used to splice the image in the Image Splicing stage.

We will use a custom algorithm (which uses the black-and-white version of the image):

1. To segment into lines:
    (a) Scan rows of pixels until you find a row with some pixels on.
    (b) Keep scanning until you find an empty row.
    (c) Everything between those is one line.
2. To segment those lines into words:
    (a) Within a line, scan vertical columns of pixels until you find a column with some pixels.
    (b) Keep scanning until you find an empty column.
    (c) scan for another column with pixels in it.
    (d) If the distance between those is above some threshold, interpret it as a space.

Try making the threshold proportional to the typical line height, to adjust for writing size.

4.3 User Verification

An optional stage, this allows the user to correct any layout detection errors from the first stage. It is to take the Source image and the previously generated SegmentLayout as its input. As output it will return the input SegmentLayout which may or may not have been modified by the user.

4.4 Image Splicing

This stage takes the same input as the User Verification stage. That is, the Source image and the SegmentLayout (either modified by the User Verification stage or straight from the Layout Detection stage). Its output is the input SegmentLayout with the segment bitmaps filled in by splicing the image using the specified co-ordinates.

Chapter 5

Processing

5.1 Training

We will need to build a collection of training data and test data. This will make it possible to automatically test the accuracy of new or modified algorithms. For word-by-word methods, the training data and test data will need to include many of the same words.

It will be necessary to be able to store pipeline results. Especially results that receive manual feedback, so that this wouldn't be necessary every time.

It should be possible to save pipelines and sets of pipelines as well.

It may be much more convenient to be able to store internal classifier data (where applicable). Brute-force methods will still need to store the features detected during training.

The pipeline for training saves the features detected to the datastore. The pipeline for classification passes the features to the classifier; the brute-force classifier (DifferencesClassifier) compares them to all of the examples in the datastore (which can be from several images).

5.2 Processing Algorithms

There are many modules that detect various features (specified as algorithms, rather than learnt). Each of these has a difference measure, that specifies how different it should regard two word images based on those features.

The program also detects width and height as another feature of the image.

The final module is a DifferenceClassifier, which combines the difference measures from several other modules, and does a brute-force search for the word in the dictionary which has the least difference to the word currently being classified.

5.2.1 FindDots

Scans the image pixel by pixel. When a black pixel is found it is added to a new blob or added to an existing blob if it is adjacent to that blob. Any adjacent blobs are then joined. When the scan has finished, any blob above a minimum size and below a maximum size is added as a 'dot'.

Any dots found that are close to the right hand side of the image are classified as full stops rather than dots (such as on 'i's and 'j's).

5.2.2 FindLines

Segments the image into rectangles. For each rectangle, if the total number of black pixels in the rectangle are above a certain threshold it is added to the start of a line. It continues scanning and adds any adjacent blocks to the line if their area is above the threshold. For the resulting lines, any that have length above a threshold are added to the list of lines.

5.2.3 FindAreaBlocks

5.2.4 Dimensions

5.2.5 OutsideArea

Finds whitespace areas around the edge of a word. Produces a set of rectangles of at least a minimum height and width. It scans along the top and bottom edges to find a place where it can fit such a rectangle, then extends it as far as possible to the right, then extends the result as far as possible upwards.

5.2.6 DifferencesClassifier

Adds up the difference measures for each module. Does a brute-force search to find the word in the training data with features least different to the current word, and guesses that.

Chapter 6

User Interface

The GUI will evolve over the course of the application development, and we will add things as we require them, this is merely a base specification.

6.1 Main GUI

The main GUI should allow the user to select the input image (preferably a page of handwritten text) and then allow the user to select the algorithms that are run. Once this is done the user should be able to select if they would like to be allow to change the image segment boundaries, as mentioned in the Image Segmentation chapter. This should allow the user to change the boundaries for each word visually, by modifying the box surrounding the word. In the case of multiple input images, this should be done before running the processing algorithms. The use should also be presented with feedback while the program is processing the handwriting, this is done through the use of classes which implement the IReporter class.

6.2 Training GUI

The user can select the input image(s), and text file(s) containing the same text as in the image(s). The user would select one or more pipelines, which would be run on this data; processing algorithms would run normally, and classifiers would run their training algorithms on the data, with the text files providing the classes for classification.

There should be a GUI section for training, which allows specifying multiple separate pipelines for comparison.

6.3 Testing GUI

The testing GUI should allow the user to specify an image and a text file like with training. The user would be able to specify multiple pipelines (algorithm congurations). It should be possible to save and load pipelines and groups of pipelines. The testing system should automatically re-run training (or parts of training) when algorithms have been updated or added. The appropriate sequence would be this:

1. The user specifies multiple pipelines
2. The user specifies the image and text file
3. The user is allowed to correct the image segmentation
4. All of the algorithms combinations are run, separately, with output about progress (and preferably also information about which features they detected etc, displayed visually if appropriate).
5. The program displays an accuracy comparison for each pipeline

Chapter 7

Data store

The data store stores information about known words. This includes sample images, features detected, word frequency information, etc. The DataStore interface will attempt to abstract away how everything is actually stored so that the initial filesystem-based storage can be replaced at a later date by something like serialization should it be required.

7.1 Stored Data

An item of data (DataItem class) in the Data Store will contain the following:

1. A unique ID (a number)
2. Zero or more sample images
3. Data such as features detected, etc, in string form.

7.1.1 ID

The Unique ID will just be a number starting at 0. Gaps will occur if items are removed from the data store. Any such gaps must be handled in such a way that iteration using something like a for loop would not encounter them. How this is done is up to the implementation.

One such way would be to fill gaps with "blank" DataItems which would also be in an internal list of unallocated items. When a new item is added to the data store rather than creating the DataItem at $max + 1$ it would just pick the first one from the list of gaps.

This way a remove would generate a gap and an add would remove it:

Items:	0	1	2	3	4
Start	allocated	allocated	allocated	allocated	allocated
Remove Item 2	allocated	allocated	unallocated	allocated	allocated
Remove item 0	unallocated	allocated	unallocated	allocated	allocated
Add new item	allocated	allocated	unallocated	allocated	allocated
Add new item	allocated	allocated	allocated	allocated	allocated

7.2 Storage

The initial implementation of the Data Store will use a directory-based layout to allow easy debugging during initial development. In this initial storage design everything will be sorted into subdirectories under one Data Store "repository" directory. Each directory under the repository directory will represent one data item:

1. The name of the subdirectory would be the items unique ID/number
2. The sample images would be stored as files inside this directory with a name like "sample-000.png"
3. Plain Text data would go in a file like "data.ini" which would have a key-value structure.

The implementation could also store anything it needs to remember inside its own data files directly inside the repository directory. How this is done (if it is done at all) is up to the implementation.

At a later date this could be serialized into a single file inside the repository directory instead.

7.3 DataStore

Access to DataItems would be through the DataStore class. The DataStore class would essentially be nothing more than a wrapper around a list of DataItems. Items would be added/retrieved/removed by unique ID just as with any other list.

7.3.1 Loading

When instructed to load the DataStore would search for saved data items. For each data item it finds it would create an instance of DataItem, instruct it to load the saved data item and then add it to its internal list. This means that it would really be DataItem that is doing most of the work - DataStores load function should be quite easy to implement

7.3.2 Saving

A save method is provided. What this does (if anything) is up to the implementation. The filesystem based data storage method discussed earlier is unlikely to require this but any future serialization based implementations may require it.

7.3.3 Interface IDataStore

```
package comp314.interfaces;

import java.util.ArrayList;

/** Data Store Interface.
```

```
 *
 * @author David Goodwin
 */
public interface IDataStore {

    /** This loads the DataStore from a specified repository.
     * @param location The location of the DataStore
     * repository
     */
    void load(String location);

    /** This saves anything required. Up to the
     * implementation.
     *
     */
    void save();

    /** Gets the item with the specified number. Gaps
     * are to  be handled by implementation in a
     * transparent way.
     * @param number The item number to get
     * @return The specified item
     */
    IDataItem getItem(int number);

    /** Adds the specified DataItem to the DataStore.
     * This should fill the first available gap if the
     * implementation allows gaps. Otherwise it should
     * be  added at count+1
     * @param theItem the Item to add
     * @throws Exception
     */
    void addItem(IDataItem theItem);

    /** Removes the specified item from the DataStore.
     * This  may either create a gap or cause every
     * item after the  specified number to be
     * renumbered. This is up to the implementation.
     * @param number The item number to remove
     */
    void removeItem(int number);

    /** Returns the number of DataItems in the
     * repository.
     * @return The number of items in the DataStore.
     */
    int getItemCount();

    /**
```

```
     * Checks if a word exists in the datastore and if so return the index.
     * If it doesnt exist return -1.
     * @param key The word to find.
     * @return The index of the word.
     */
    IDataItem exists(String key);

    /**
     *
     * @return List of all the unique words in teh store.
     */
    ArrayList<String> listWords();
}
```

7.4 DataItem

The DataItem class is the container for an item of data that is stored in the Data Store. It is in charge of providing access to that item of data including loading and saving it.

7.4.1 Loading

The DataItem is to be given a location from which to load itself. What exactly is done when this is called is up to the implementation but it is expected that once the call has been completed the DataStore should be ready for use. In the previously discussed filesystem implementation it would scan its directory for images to obtain a count and make the data file ready for use.

7.4.2 Saving

What is done when this is called is entirely up to the implementation. Once this call has been completed it is expected that the instance could be deleted without any loss of information.

7.4.3 Access to Data

Sample images are to be accessed by number using get, add and remove methods. Plain text data are to be accessed by key using get/set/remove methods.

7.4.4 Interface IDataItem

```
package comp314.interfaces;

import java.util.ArrayList;

/** Data Store Interface.
 *
```

```
 * @author David Goodwin
 */
public interface IDataItem {
    /** Loads the DataItem from the specified location.
     * @param location The location to load from
     */
    void load(String location);

    /** Saves the DataItem. The location is obtained
     * from the result of a previous save(String) or
     * load(String) operation.
     */
    void save();

    /** Saves the DataItem to the specified location.
     * @param location The location to save to.
     */
    void save(String location);

    /** Gets the specified sample image.
     * @param number The number of the sample image.
     * @return The requested sample image
     */
    IData getImage(int number);

    /** Adds a sample image.
     * @param theImage The sample image to add
     */
    void addImage(ITextImage theImage);

    /** Removes the specified image.
     * @param number The image to remove
     */
    void removeImage(int number);

    /** Returns the number of sample images.
     * @return The number of sample images
     */
    int getImageCount();

    /** Gets a stored value. Key is the key under
     * which the data was previously saved. The
     * default value is what is returned if the key
     * does not exist. This means that there is no need
     * to check the key exists first.
     * @param key The items key. Keys are in a tree. For
     * example, "features/featureA" or "ASCIIvalue"
     * @param sDefaultValue What to return if the key
     * doesn't exist.
     * @return The value for the specified key if it
```

```
     * exists, otherwise default_value.
     */
    String getValue(String key, String sDefaultValue);

    /** Stores the specified value under the specified
     * key.
     * @param key The key to save the value under
     * @param value The value to store
     */
    void setValue(String key, String value);

    /** Removes the specified key.
     * @param key The key to remove.
     */
    void removeKey(String key);

    /** Returns a list of all keys under the specified
     * key.
     * @param key The specified key. "" is the root key.
     * @return The list of keys under the specified key.
     */
    ArrayList<String> listKeys(String key);

    /**
     * Gets the uid of the item.
     * @return String containing the uid.
     */
    String getUid();

    /**
     * Sets the unique identifier of the item.
     * @param uid A unique string.
     */
    void setUid(String uid);
}
```

7.4.5 Reference Keys

Key	**Explanation**
ascii_word	The text representation of the word.
`outsideArea\<edge_id>\<rectangle_id>\x`	X Coordinate
`outsideArea\<edge_id>\<rectangle_id>\w`	Width
`outsideArea\<edge_id>\<rectangle_id>\y`	y Coordinate
`outsideArea\<edge_id>\<rectangle_id>\h`	Height.

Value Keys

Keys for the setValue/getValue/removeKey/listKeys methods are in a tree form. For the string “key1/key2”, “key2” is a child node of “key1”. listKeys(“key1”) should return the list including “key2”. listKeys(“”) should return the list including “key1”

Chapter 8

Output

8.1 Logging

The logging should be a static class which is available to all modules and classes, and provides the ability to write to the log file, which should be opened upon program execution and closed upon termination.

8.1.1 Severity

Each log entry will contain the entry severity at the start. The writeLine(String) function uses the "Notice" severity where as the writeLine(Severity,String) allows you to specify the severity.

Available Severity levels

1. Notice - Just a general notice.
2. Debug - A debug message - There shouldnt be too many of these.
3. Warning - A warning that the program state is not quite as expected and something may go wrong later because of it.
4. Error - Something has gone wrong but recovery may be possbile. The program will attempt to continue but a Failure may later occur.
5. Failure - An unrecoverable error has occured. This message should probably include a stack trace and any program state information. After issuing one of these the program should try to terminate nicely.

8.1.2 ILogger

```
package comp314.interfaces;

import java.io.IOException;

/** Interface for logging classes.
 *
```

```
 * @author David Goodwin, Daniel Buchanan
 */
public interface ILogger {

    /** The log entries severity level.
     *
     */
    public enum Severity {Notice, Debug, Warning, Error,
        Failure};

    /** Opens the log file for input.
     * @param filename The output file for logging.
     */
    void open(String filename) throws IOException;

    /** Closes any open log files.
     */
    void close() throws IOException;

    /** Writes a message to the log file using the Notice
         severity.
     * @param message The message to write to the log
         file.
     */
    void writeLine(String message);

    /** Writes an entry to the log file using the
        specified severity. This can
     * throw exceptions
     * @param sv Log entry severity
     * @param message Log message
     */
    void writeLineE(Severity sv, String message) throws
        IOException;

    /** Performs the same job as WriteLineE except does
        not throw any exceptions.
     *
     * @param sv Severity Level
     * @param message Log Message
     */
    void writeLine(Severity sv, String message);

    /** Returns the String representation of the
        specified Severity Level. This
     * is the String representation used by the log
         output.
     * @param sv The severity level
     * @return The string representation of the given
         Severity Level.
```

```
     */
    String svToString(Severity sv);
}
```

8.2 Program Output

8.2.1 Training

Will output statistics to the log file on each run.

8.2.2 Normal Mode

Will output a text file containing the recognised contents of the input, wherever the user decides to put it.

8.2.3 Testing Mode

Will output statistics etc to the log file on each run.