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.

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.

```
1 package comp314.interfaces;
2
3 /**
4  *
5  * @author danielbuchanan
6  *
7  */
```

```
 8 public interface IPipelineModule {
 9     /**
10      *
11      * @param data
12      */
13     void setData(IData data);
14
15     /**
16      *
17      * @return IData
18      */
19     IData run();
20
21     /**
22      *
23      * @param data
24      * @return IData
25      */
26     IData run(IData data);
27
28     /**
29      *
30      * @param reporter
31      */
32     void setReporter(IReporter reporter);
33
34     /**
35      *
36      * @return IData
37      */
38     IData getData();
39
40     /**
41      *
42      * @return IData
43      */
44     IData getResult();
45
46     /**
47      *
48      * @param options
49      */
50     void setOptions(IData options);
51 }
```

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.

```
1 package comp314.interfaces;
2
3 /**
4  *
5  * @author danielbuchanan (replace with whoever actually
       created this)
6  *
```

```
7  */
8  public interface IReporter {
9      /**
10      *
11      * @param status
12      */
13     void setStatus(String status);
14
15     /**
16      *
17      * @param progress
18      */
19     void setProgress(int progress);
20 }
```

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).

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 Noise Reduction

This optional module will take an input bitmap and return a noise-reduced bitmap. It should check what type of noise is affecting the input image and then choose a filter such as Median which removes the noise and returns the noise reduced image. Noise types could include:

1. Salt & pepper - Pixels in the image that are very different in colour or intensity to the surrounding pixels

2. Gaussian Noise - each pixel will be changed by a small amount from its original pixel value

Median Filter

Median filter considers each pixel in the image. For each pixel it sorts the neighbouring pixels in order based on their intensity. It then replaces the original value of the pixel with the median value from the list.

3.1.3 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. (AR to expand this)

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.

5.2 WEKA integration

There will be pipeline modules that provide wrappers to WEKA classification algorithms. These modules will automatically develop a suitable set of WEKA Attributes based on features detected earlier in the pipeline.

There could be a module to provide a wrapper for WEKA clustering algorithms. This would detect similarities between instances; cluster membership could then be regarded as like another feature and be sent to the classifiers.

We may find the WEKA GenericObjectEditor useful for configuring pipeline modules. This provides an automatically-generated dialog box with a specified set of options.

5.3 Processing Algorithms

All algorithms take in a binary bitmap image, and produce a list of word guesses (Strings), as well as a ranking. (Could make use of a Tuple class here).

Require a module to be put at the end of the pipeline to pull together all results and output in a set format.

Modules to recognise to several different features, making use of differing learning algorithms.

How to move data around? If identifying individual features, pass around list of found features.

The initial processing algorithms should be:

1. The area rectangles around the outside of a word
2. nearly horizontal or vertical lines, and crossing lines
3. dots (such as in i j . ? !)

There should also be a module (or possibly a facility, usable by several modules) that keeps track of word frequency. This would include an initial database of word frequencies, and be able to update word frequency information based on the user's input.

Also, there could be slightly normalized measures of the bitmap width of the word, which is related to the number of characters.

There could be a normalized measure of word height, but it would be more appropriate to discover which vertical sections the word has. All words have the middle section, if they have letters that have lines that go up or down then they have those sections. For example, "panda" - the 'p' is in the middle and lower section, and the 'd' is in the middle and higher section. Also, most capital letters include the top section. This measure could be discovered based on simply whether the word has pixels near the top or bottom of the line, or by detecting the actual parts of those letters that take more than one section.

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

```
1 package comp314.interfaces;
2
3 /** Data Store Interface.
4  *
5  * @author David Goodwin
```

```
6   */
7   public interface IDataStore {
8
9       /** This loads the DataStore from a specified
            repository.
10      * @param location The location of the DataStore
11      * repository
12      */
13      void load(String location);
14
15      /** This saves anything required. Up to the
16      * implementation.
17      *
18      */
19      void save();
20
21      /** Gets the item with the specified number. Gaps
22      * are to  be handled by implementation in a
23      * transparent way.
24      * @param number The item number to get
25      * @return The specified item
26      */
27      IDataItem getItem(int number);
28
29      /** Adds the specified DataItem to the DataStore.
30      * This should fill the first available gap if the
31      * implementation allows gaps. Otherwise it should
32      * be  added at count+1
33      * @param theItem the Item to add
34      * @throws Exception
35      */
36      void addItem(IDataItem theItem) throws Exception;
37
38      /** Removes the specified item from the DataStore.
39      * This  may either create a gap or cause every
40      * item after the  specified number to be
41      * renumbered. This is up to the implementation.
42      * @param number The item number to remove
43      */
44      void removeItem(int number);
45
46      /** Returns the number of DataItems in the
47      * repository.
48      * @return The number of items in the DataStore.
49      */
50      int getItemCount();
51  }
```

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

```
1  package comp314.interfaces;
2
3  import java.util.List;
4
5  /** Data Store Interface.
6   *
7   * @author David Goodwin
8   */
9  public interface IDataItem {
10     /** Loads the DataItem from the specified location.
11      * @param location The location to load from
12      */
13     void load(String location);
14
15     /** Saves the DataItem. The location is obtained
16      * from the result of a previous save(String) or
17      * load(String) operation.
18      */
19     void save();
20
21     /** Saves the DataItem to the specified location.
22      * @param location The location to save to.
23      */
```

```
24      void save(String location);
25
26      /** Gets the specified sample image.
27       * @param number The number of the sample image.
28       * @return The requested sample image
29       */
30      IData getImage(int number);
31
32      /** Adds a sample image.
33       * @param theImage The sample image to add
34       */
35      void addImage(IData theImage);
36
37      /** Removes the specified image.
38       * @param number The image to remove
39       */
40      void removeImage(int number);
41
42      /** Returns the number of sample images.
43       * @return The number of sample images
44       */
45      int getImageCount();
46
47      /** Gets a stored value. Key is the key under
48       * which the data was previously saved. The
49       * default value is what is returned if the key
50       * does not exist. This means that the keys existance
51       * does not need to be checked first.
52       * @param key The items key. Keys are in a tree. For
53       * example, "features/featureA" or "ASCIIvalue"
54       * @param sDefaultValue What to return if the key
55       * doesnt exist.
56       * @return The value for the specified key if it
57       * exists, otherwise default_value.
58       */
59      String getValue(String key, String sDefaultValue);
60
61      /** Stores the specified value under the specified
62       * key.
63       * @param key The key to save the value under
64       * @param value The value to store
65       */
66      void setValue(String key, String value);
67
68      /** Removes the specified key.
69       * @param key The key to remove.
70       */
71      void removeKey(String key);
72
73      /** Returns a list of all keys under the specified
```

```
74          * key.
75          * @param key The specified key. "" is the root key.
76          * @return The list of keys under the specified key.
77          */
78         List<String> listKeys(String key);
79 }
```

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

```
1 package comp314.interfaces;
2
3 import java.io.IOException;
4
5 /** Logging interface.
6  *
```

```
7  * @author Daniel Buchanan, David Goodwin
8  *
9  */
10 public interface ILogger {
11
12     public enum Severity {Notice, Debug, Warning, Error,
           Failure};
13     /** Opens the logfile for input.
14      * @param path The output file for logging.
15      */
16     void open(String filename) throws IOException,
           Exception;
17
18     /** Closes any open logfiles.
19      */
20     void close() throws IOException, Exception;
21
22     /** Writes a message to the logfile using the Notice
           severity.
23      * @param message The message to write to the logfile
            .
24      */
25     void writeLine(String message);
26
27     /** Writes an entry to the logfile using the
           specified severity. This can
28      * throw exceptions
29      * @param sv Log entry severity
30      * @param message Log message
31      */
32     void writeLineE(Severity sv, String message) throws
           Exception;
33
34     void writeLine(Severity sv, String message);
35
36     /** Returns the String representation of the
           specified Severity Level. This
37      * is the String representation used by the log
            output.
38      * @param sv The severity level
39      * @return The string representation of the given
            Severity Level.
40      */
41     String svToString(Severity sv);
42 }
```

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.