GestureLab User and Developer Documentation

Adrian Bickerstaffe and Victoria Colquhoun
School of Information Technology
Monash University
Clayton, Victoria 3800
Australia

September 30, 2008

Contents

1 Introduction

GestureLab is a software tool designed to facilitate rapid development and testing of domain specific gesture recognizers. Recognizers can be developed entirely within GestureLab without any need for a testbed application and can be coupled with grammar engines such as Cider [1] without modification.

GestureLab recognizers follow the standard approach to statistical gesture recognition: recognition is performed on digital ink which includes position, timing, pressure, and angle data. Statistical summary features such as the total length of the gesture, initial stroke angle, and maximum curvature are extracted from this data and used by a classifier algorithm to predict class labels (gesture types). A recognizer thus consists of a bundle of feature extractors and a classifier algorithm trained on a particular gesture corpus.

GestureLab supports all phases of the recognizer development process: (a) collecting, manipulating and sharing gesture corpora, and (b) automatic training and cross-validation of feature extraction and recognizer mechanisms. In the event that the built-in feature extraction and recognizer mechanisms are insufficient, GestureLab also allows the developer to readily define (c) new feature extraction mechanisms and (d) new recognizer algorithms. GestureLab generates probabilistic recognizers that return membership probabilities for all possible token classes instead of a single most likely class. Probabilistic classificators can be useful during disambiguation at a syntactic level.

2 User Manual

2.1 Software Installation

GestureLab has been developed for Apple Macintosh ("Mac") computers running Mac OSX 10.4 or later. The software package is available in the public domain at `http://www.csse.monash.edu.au/~berndm/gesture_lab/GL/`. This build is universal, meaning that GestureLab will run on Macs which are based upon either PowerPC or Intel hardware. The `GestureLab-1.0a4.zip` archive contains the program proper, a default feature extractor plugin which implements the features of [4], a default recognizer plugin which provides DAG-SVM [3] and MCST-SVM recognizers [2], and an example database containing a large corpus of gestures.

To install GestureLab, first extract the contents of `GestureLab-1.0a4.zip`. The main `GestureLab` executable can reside anywhere on disk and is most commonly installed in a user's home directory or some space common to all users (e.g. `/Applications/`). The application contains the default feature extractor and recognizer plugins and therefore nothing must be done in regards to these plugins. Next, create a directory `~/Library/Digest/` and move the GestureLab database file, `database.dat`, into this location.[1] New recognizers created by GestureLab will also be written to the `~/Library/Digest/` directory. Be sure to set the appropriate file permissions of GestureLab, `~/Library/Digest/` and the `database.dat` file contained within.

Modern Mac processors run GestureLab at acceptable speeds for most tasks, however 1Gb or more of main memory is advised. However at the time of writing, the speed and responsiveness of GestureLab can be lacking when information about large numbers of gestures must be retrieved and displayed. Additionally, typical training times for the default recognizer algorithms are in the order of hours (or even days) for problems of five-way classification or more with a reasonably large number of samples per class (e.g. 100). Optimization of the software is planned as future work.

[1] Although GestureLab appears to support alternative paths for the database file, this feature is at present not working.

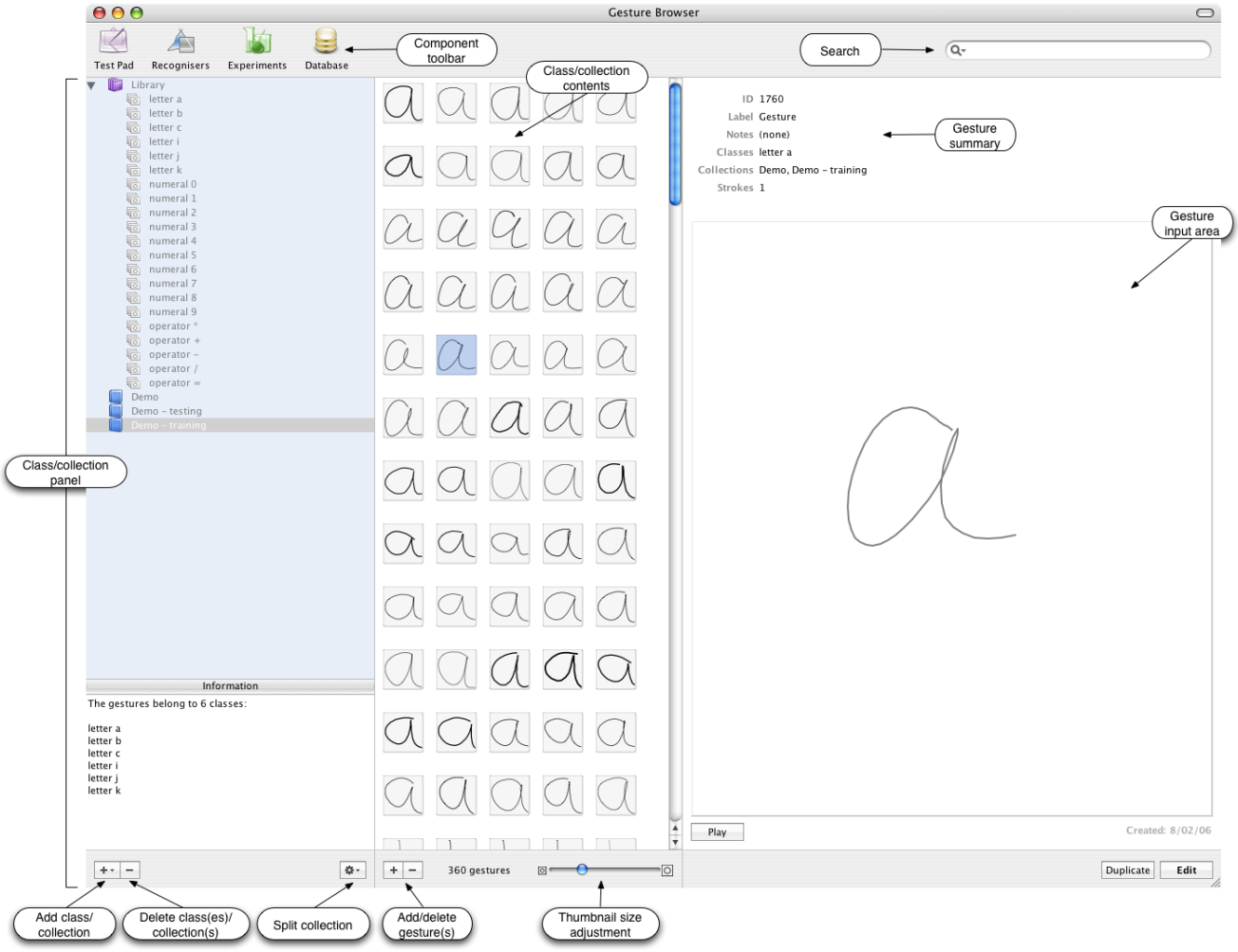

Figure 1: The main window of GestureLab.

2.2 The GestureLab Environment

The main GestureLab window comprises four areas which are discussed below.

Component toolbar Provides access the the test pad, recogniser training wizard, experiment browser and wizard, and database browser (see Section 2.3).

Class/collection panel GestureLab arranges gestures in terms of a library containing named *categories* (or "classes") of gestures and *collections* of gestures selected from these categories. Class membership determines the intended interpretation of a gesture, while collections are named sets used during training and testing. Each gesture may belong to any number of named collections and classes. In this way, training and test collections can be created, modified, and deleted without altering the corpora. The library can be accessed and manipulated using an intuitive drag-and-drop interface or via an SQL interface (see Section 2.3.4).

Click and hold the button labelled + to create new empty classes and collections. Clicking the "Split collection" button invokes a dialog box for splitting the currently selected collection randomly to create new training and testing collections. The training/testing portions default to 60%/40%, however these portions can be adjusted by the user. The name of each new collection is formed

from the name of the source class/collection with the added suffix `-training` or `-testing`. Click the button labelled `-` to delete the currently selected classes and/or collections (after confirmation). Note that deleting collections will *not* remove the gestures from the class(es) to which they belong.

The "Information" panel lists the all classes to which the gestures in the selected collection belong. This information is useful to obtain a quick summary of the alphabet covered by a collection. Whilst the same type of information is displayed when a particular class is selected, this information is not very useful since all gestures must belong to that single class.

Class/collection contents This area displays a scrollable list of thumbnails which represent the gestures of the selected class or collection. The size of the thumbnails can be adjusted using the slider control, noting that adjusting the size of the thumbnails does not in any way scale or alter the gestures' stroke data. Clicking the button labelled `+` will create a new blank gesture which by default belongs to the currently selected class.

Thumbnails are selected using individual mouse-clicks or a click-and-drag approach. Alternatively, typing `Apple-a` when the class/collection contents pane is active will result in all gestures being selected. Selected gestures can be assigned to classes and collections by dragging the selection over a class/collection entry in the class/collection summary pane. Clicking the button labelled `-` will delete all currently selected gestures (after confirmation). Again, gestures that are deleted from a collection will not be deleted from the class(es) to which they belong.

Gesture input area and summary The main role of this pane is to provide an area for gesture input. To input a new gesture, select a blank or existing gesture thumbnail from the "class/collection contents" area and click the "Edit" button. The input area will now become active and a gesture can be drawn in this area and, if required, the input gesture can be redrawn once the multi-stroke timeout has elapsed. The input process is finalized by clicking the "Done" button. The "Duplicate" button creates an exact copy of the currently selected gesture including its stroke data, class/collection membership and notes (but not its unique identifier number). Gesture duplication is typically used to quickly add new gestures (which will be edited later) with class and collection membership that matches an existing gesture, thus avoiding having to set class and/or collection membership manually for each new gesture.

The "Play" button animates in the input area the currently selected gesture. The animation redraws the existing gesture exactly as it was drawn, including stroke directions, speed, and pauses between strokes in a multi-stroke gesture. This software feature is particularly useful for examining visually why particular gestures are mis-classified in an experimental trial (see Section 2.3.2).

Finally, the upper region of the gesture input area provides information about the currently selected gesture including a unique identifier number, classes and collections to which the gesture belongs, and any notes recorded for the gesture during the input stage.

Search This is currently an unimplemented feature.

2.3 Components

There are four main components which may be invoked by clicking icons on the component toolbar: the recognizer training wizard, the experiment browser and testing wizard, the test pad, and the database browser. Each of these components is now discussed in detail.

2.3.1 Recognizer Browser

The recognizer browser provides a summary of trained recognizers that are recorded in the GestureLab database (see Section 2.3.4), noting that GestureLab is distributed without any trained recognizers by default. Figure 2 shows that selecting a trained recognizer provides information about that recognizer such as the date of training, the model file which stores the recognizer in `~/Library/Digest`, along with the feature set and training corpus used to train the recognizer. Clicking the "Training set" hyperlink will create in the main window a temporary collection which contains all gestures used to train the

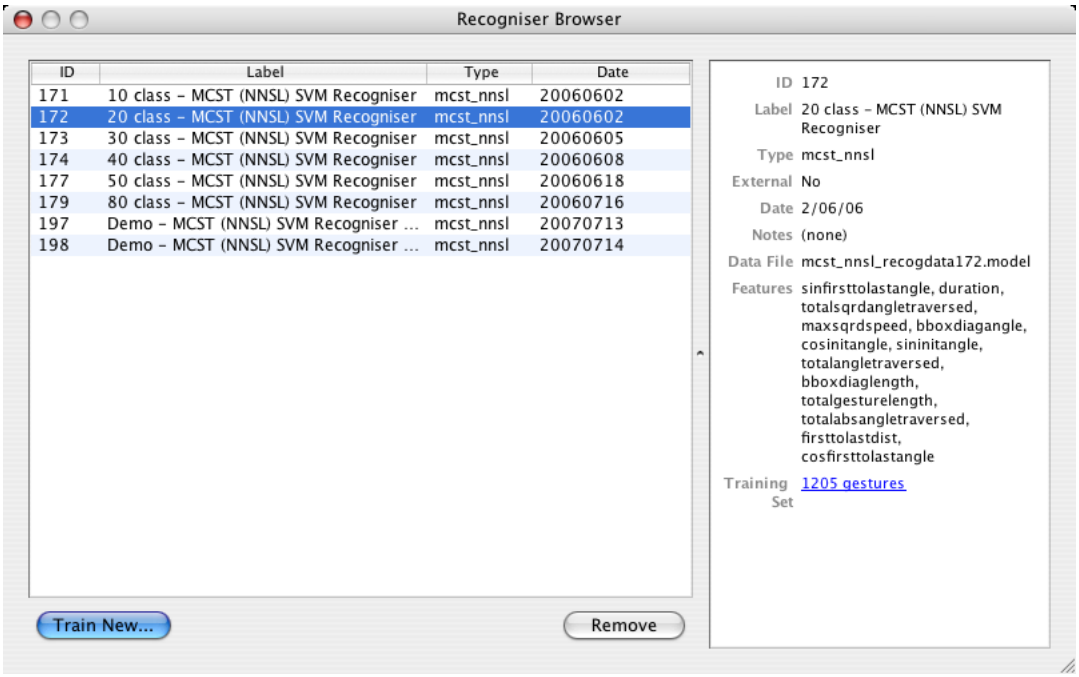

Figure 2: The recognizer browser.

currently selected recognizer. This collection is useful to check quickly which classes are predicted by the recognizer, and may also reveal why the recognizer is misclassifying certain types of test gesture, e.g. the recognizer was trained with only right-handed input and misclassifies gestures drawn by a left-handed person. Clicking the "Remove" button will delete the currently selected recognizer the GestureLab database. Note that this operation removes the record of the recognizer from the GestureLab database but it does not delete the respective `.model` file from `~/Library/Digest`.

The recognizer browser also provides an assistant (or "wizard") which guides the user through the process of training a new recognizer in a step-by-step manner. Assuming the "Train New" button is clicked, recognizer training is configured over five steps as follows:

1. Select the recognizer algorithm from a drop-down list of available recognizers. The standard recognizer plugin provides a DAG-SVM implementation and four varieties of the MCST-SVM classifier, however additional plugins may contribute to the list of available recognizer algorithms. The nearest neighbour single-link (NNSL) MCST-SVM recognizer provides high performance for generic corpora and is therefore a reasonable choice for preliminary experiments.

2. Select the feature set that will be extracted from gestures and used to train the recognizer (and later make test predictions). The standard feature plugin implements the 13 features detailed in [4] and these features are selected for use by default. The features provided by any additional feature plugins are also selected for use automatically. Enabled features are shown in the right-hand pane shown in Figure 3; unused features are shown in the left-hand area. Individual features are activated and deactivated by selecting and dragging the feature name from the right-hand side to left-hand side and vise-versa. The *order* the features extracted can be altered by dragging the entry of a particular feature up or down the list of enabled features. **Important note:** GestureLab will crash during training if no features are selected.

3. Select the training corpus of gestures from the list of classes and collections. A training corpus can be formed from the gestures of one or more classes, and/or one or more collections. Whilst the corpus can be formed by selecting several classes, the more typical approach is to create a training collection prior to the start of training and select the collection at this step of the training assistant.

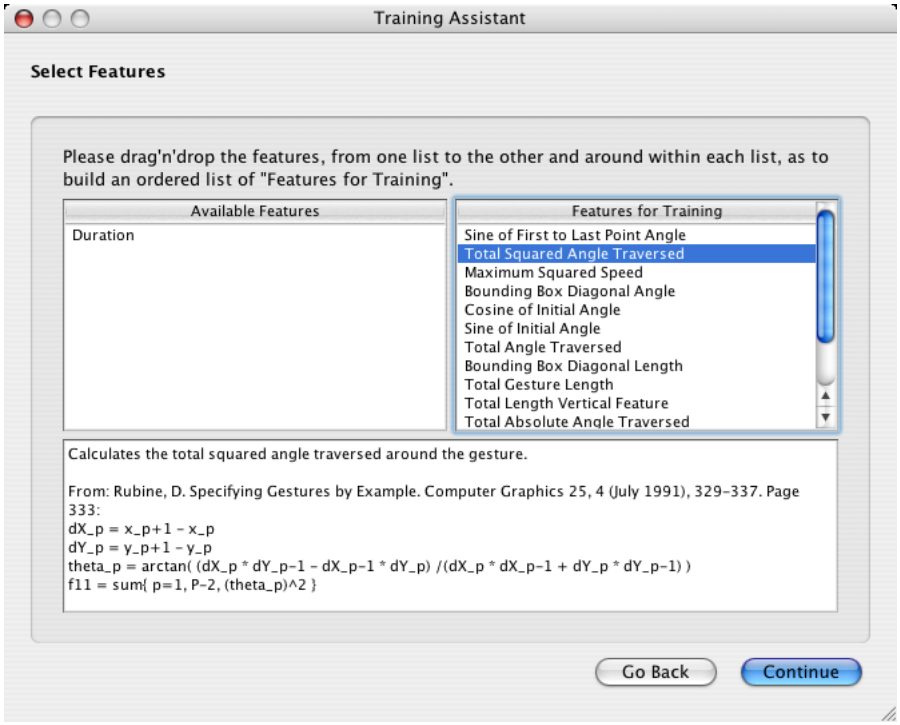

Figure 3: Training feature selection.

In this way, the training collection can be labelled explicitly and used to train other recognizers at a later stage.

4. Enter a short textual label that will be used to identify the trained recognizer to the user at a later stage. By default, this label indicates the type of recognizer algorithm that has been selected for training, however this text can be edited. More verbose notes about the recognizer can also be added, where example notes may include the name of the person who trained the recognizer, the reason for training the recognizer, etc. The ability to enter recognizer parameters is at present an unimplemented feature of GestureLab.

5. The final stage of the training process is to confirm the options selected in Steps 1 to 4. Confirmation is based upon a summary of training details including the chosen recognizer algorithm, feature set, and the corpus of gestures selected for training. If any of these details require modification, the "Back" button provides a means to return to previous stages of the training assistant. Training is commenced by clicking the "Train" button; a progress bar of training progress will then be displayed. Note that the progress bar reflects the percentage of gestures for which features have been extracted; it does *not* represent the actual progress of training the classifier algorithm. This means that GestureLab may appear to have crashed or become caught in a loop during training when this is not the case. Training the default recognizers typically requires minutes, hours or even days to complete for problems of three-way classification or higher using corpora in the order of 100s or 1000s of gestures. The default recognizers report training progress in the system console.

Once trained, a recognizer is *independent* of its training data; the recognizer can be used in external applications to classify gestures and no reference to the training data must be made.

2.3.2 Experiment Browser

The experiment browser manages the results of experiments designed to test the performance of recognizers trained by GestureLab (see Section 2.3.1). Figure 4 illustrates the experiment browser window.

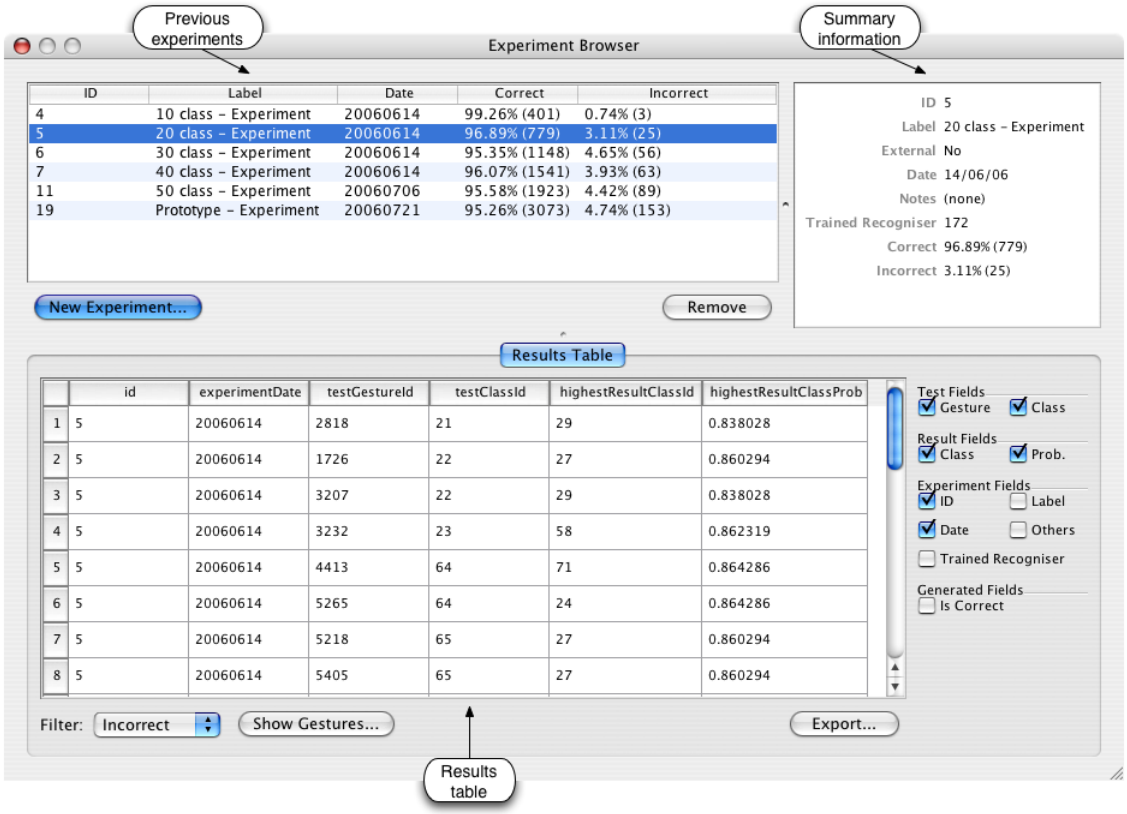

Figure 4: The experiment browser.

Previous experiments This area lists all previous experiments recorded by the GestureLab database (see Section 2.3.4). Each entry in the list includes a textual label given to the experiment, the date of execution, along with a top-level summary of recognizer performance. Performance is reported in terms of the percentages of correct and incorrect predictions across the test corpus; the number of correctly/incorrectly classified gestures is shown in parenthesis. Clicking "New Experiment" will invoke the experiment assistant (see below), whilst clicking "Remove" will remove the currently selected experiment from the GestureLab database.

Summary information This pane provides a neater summary of the currently selected experiment. In addition to the information provided by the "Previous experiments" panel, the summary information area includes any notes recorded for the experiment and the identifier number of the trained recognizer used to perform the experiment.

Results table The results table provides a flexible way to view detailed information about the currently selected experiment. The particular columns of the table that are displayed can be controlled using the "Test Fields" check boxes shown to the right of the dynamically generated table. The "testClassId" column lists the class(es) to which each gesture is known to belong; "highestResultClassId" lists the class that received the highest probability prediction by the recognizer; and "highestResultClassProb" provides the probability of the most likely class as predicted by the recognizer.

A filter can be applied so that the table displays results for all test gestures, only gestures classified correctly, or only gestures classified incorrectly. The latter filters are particularly useful when used in conjunction with the "Show Gestures" feature; clicking "Show Gestures" creates a temporary collection containing all gestures currently displayed in the table. In this way, the user can inspect visually all gestures that are misclassified and look for traits in the gestures which may have caused them to be misclassified. Likewise, the user can create a temporary collection of all correctly classified gestures and compare the shape and animation of these samples to misclassified examples of the same class.

Note that the table "Export" function is unimplemented at the time of writing.

The experiment browser also provides an "Experiment Assistant" which guides the user through the process of configuring and executing a new experiment. This process consists of the following steps:

1. Select the trained recognizers to be tested; these recognizers are trained using the process described in Section 2.3.1. Note that one experiment record will be created for each recognizer being tested and that features need not be specified for experiments since GestureLab records the feature set used to train each recognizer. These features are extracted automatically during test classifications.

2. Select a testing corpus for the experiment. The testing corpus consists of one or more gesture classes/collections which are created and shown in the main GestureLab window. The testing corpora is most often a collection created automatically using the random split feature described in Section 2.2.

3. Enter a short label for the experiment. This label will appear in the experiment browser once the experiment is complete and therefore the label should reflect the nature of the experiment (i.e. what is being tested). The user may also add more extended notes about the experiment at this stage. Such notes may provide information about who is undertaking the experiment, the aim and expected out comes of the experiment, and so on.

4. Confirm the options selected in Steps 1 to 3. Confirmation is based upon a summary of experiment details including the experiment label, recognizers to be tested, and the testing corpus. If any of these details need modification, the "Back" button provides a means to return to previous stages of the experiment assistant. Clicking the "Begin" button will commence the experiment. Unlike the training process, experiments complete quickly and typical running times are in the order of seconds or at most minutes.

2.3.3 Test Pad

The test pad allows recognizers to be evaluated on a gesture by gesture basis using interactive input instead of a whole gesture collection as is the case with the experiment browser. The test pad provides a quick interactive method for testing recognizers.

Figure 5 illustrates the test pad window. The "Trained Recognizers" panel lists all recognizers recorded in the GestureLab database and, using the check boxes beside each recognizer, individual recognizers can be activated and deactivated on demand. Clicking the "All" button will activate all recognizers known to GestureLab, whilst clicking the "None" button will uncheck all boxes and deactivate every recognizer. Combinations of active recognizers can be used to test the relative performance of particular recognizers quickly and perhaps evaluate whether a change in recognizer algorithm or feature set used has improved recognition performance. Below the list of recognizers is an information panel which contains details about the currently selected recognizer.

Test gestures are drawn in the main input area labelled in Figure 5 using a desktop tablet, tablet monitor, or mouse. Recognition occurs when either the multi-stroke timeout elapses or when the "Recognise" button is clicked. Note that GestureLab uses information in its database to extract features from them test gesture which match those used to train each active recognizer. The results of classification are tabulated in the panel below the input area; these results are displayed as the probability of the input stroke belonging to each of the classes for which the current classifier was trained. The most probable

classification is highlighted with bold text as Figure 5 shows. Clicking the "Clear" button will erase the last input gesture, though this occurs automatically when a new gesture is drawn over the last classified gesture.

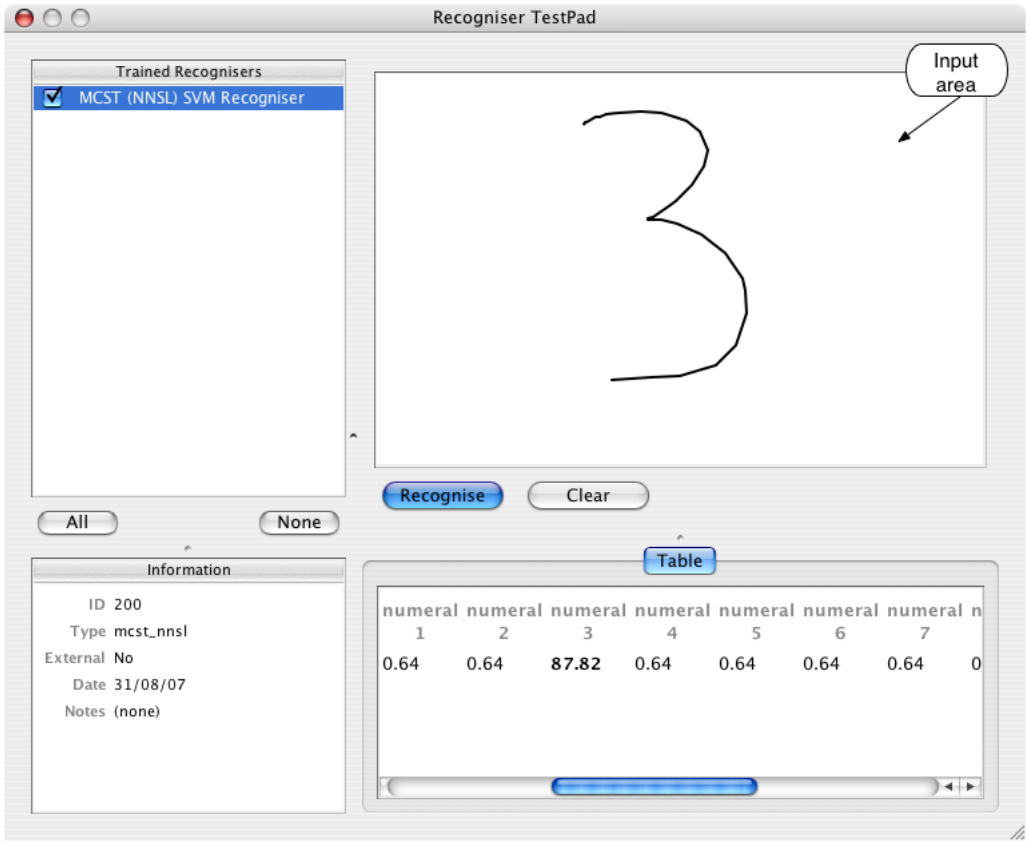

Figure 5: Test pad.

The test pad is particularly useful for investigating why certain recognizers misclassify particular gestures. Using the experiment browser (see Section 2.3.2), a temporary collection of misclassified gestures can be created and these gestures can be viewed at original size and animated. Using this information, the user may suspect certain attributes of the gestures (such as left or right handedness, or slant) are causing the misclassifications. The user can then replicate these traits with gestures drawn interactively in the test pad. Based on the results of test pad experiments, new recognizer algorithms and/or features may be designed and implemented and thus, the test pad provides a "hands on" approach to improving the performance of recognizers.

Note that other resource-intensive processes running simultaneously with GestureLab may affect the accuracy of recognizer predictions in the Test Pad component. This occurs because the capturing of stroke data is bound by processor time and consequently the stroke data of test gestures will be sampled with lower resolution when another application occupies the majority of the computer's processor time.

2.3.4 Database Browser

GestureLab stores all information about its corpora, recognizers, and experiments in a single SQL database file which is by default stored on the local machine in `~/Library/Digest/database.dat`. Remote access to this database is possible using GestureLab clients connected via the Internet. Remote database access makes it possible for geographically distributed research groups and for whole research communities to contribute to shared corpora, and to use this data for recognizer development. The

address and port of the remote server must be specified when GestureLab is first started, along with a username and password for access to the remote database.

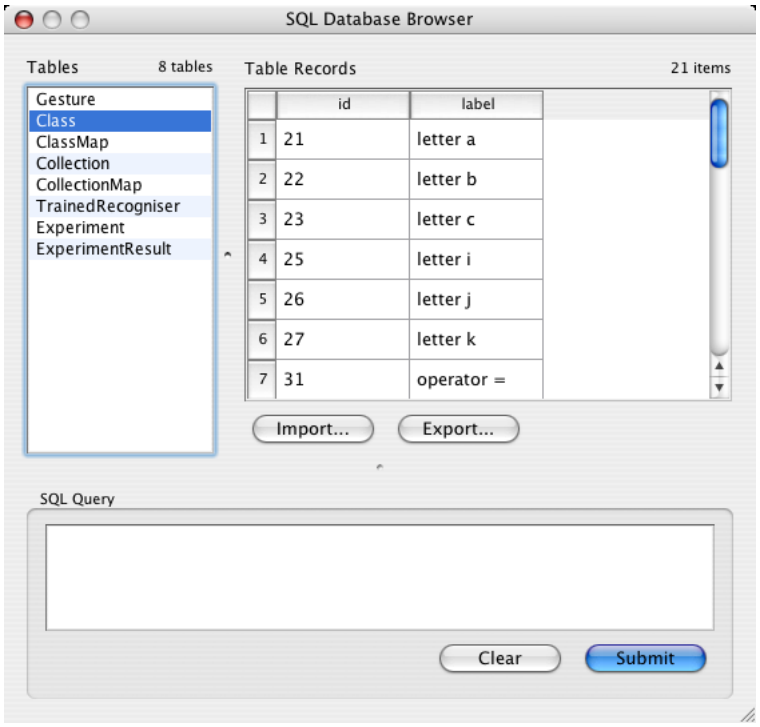

Figure 6: The database browser.

The database browser allows a user to visually inspect the database contents via a graphical interface and manually query the database using standard SQL syntax. Clicking one of the database tables listed in the browser will display the contents of that table in the adjacent window; Figure 6 shows the database browser being used to view the mapping between class names and internal class identifiers. Database contents can be edited directly by clicking on the relevant field in the "Table Records" area and entering modifications to the existing record. Contents can be added to the currently selected table using the "Import" function which accepts data files in comma-separated value (CSV) format. The import function is particularly useful for situations in which gesture data has been captured from an external source and the user wishes to add these gestures to the GestureLab database. In such cases, pre-processing of the data is usually required to obtain the required CSV format. The "Export" button will write the contents of the currently selected table to a text file in CSV format. This function is most often used to export gesture data (captured in GestureLab) for use in an external application. The "Class" table is typically exported to obtain a print-out of the mapping between class labels and the integers which identify the classes uniquely. Such information is useful when a recognizer trained by GestureLab is integrated into an external application; here, the developer needs to know which classes correspond to the integer identifiers returned by the recognizer when a prediction is made.

SQL queries are entered in the bottom input area and may apply to all tables, not just the currently selected table. The results of an SQL query will be displayed in the "Table Records" area.

3 A Tutorial on GestureLab Plugins

This section provides tutorials to develop GestureLab plugins and integrate the trained recognizers by GestureLab into external applications. The tutorials assume that the reader is reasonably well-versed in the C++ and Java programming languages.

Example feature and recognizer plugins are developed. The feature extractor class is developed in `examplefeature.h` and `examplefeature.cpp`, and this class is used to develop a GestureLab feature plugin in `examplefeatureplugin.h` and `examplefeatureplugin.cpp`. A recognizer class is developed in `svmrecogniser.h` and `svmrecogniser.cpp`. This class is used to implement a GestureLab recognizer plugin in `svmrecogniserplugin.h` and `svmrecogniserplugin.cpp`. Each plugin class implements a "factory" to create an instance of the feature extractor or recognizer classes as required by GestureLab.

3.1 An Example Feature Plugin

This tutorial will develop a GestureLab feature plugin which provides a single feature: the distance between the first and last points of a stroke. The plugin provides only one feature for the sake of brevity; a more typical feature plugin will provide a *set* of features which are bundled into a single package. Extension of the plugin to accomodate more features is trivial. Each plugin should occupy a separate directory in `digest/src/plugins` of the GestureLab source code tree; in this case, the source files of the new feature plugin are located in `digest/src/plugins/ExampleFeature`.

3.1.1 examplefeature.h

Each feature is defined by a separate C++ class and for the purposes of this tutorial, the feature class is declared in a new file `examplefeature.h`. This header file begins with protective macros before including `stlfeatureinterface.h` which defines the interface that all features must implement.

```
#ifndef EXAMPLEFEATURE_H
#define EXAMPLEFEATURE_H

#include "../../GestureRecognition/stlfeatureinterface.h"
```

The class `FirstToLastDistFeature` represents the single feature which will be provided by the plugin and is defined as follow. The `FirstToLastDistFeature` class must inherit and implement the inferface defined by `StlFeatureInterface`. The first member functions of `FirstToLastDistFeature` make use of macros defined in `StlFeatureInterface`. `DECLARE_STL_FEATURE_KEY` sets an identifier string for the feature and which is used by GestureLab internally to uniquely identify each feature. Consequently, it is essential that this key is unique for each feature. Following this, the `DECLARE_STL_FEATURE_TITLE` macro is called to set a descriptive title for the feature. This string appears in the GestureLab training component when features for training are selected and therefore the string should provide a concise description of what the feature represents in terms of gestures. Finally, `DECLARE_STL_FEATURE_DESCRIPTION_METHODS` is called to provide methods which set a more verbose textual description of the feature. This includes a wrapper function for `classDescription()` which is defined later.

```
class FirstToLastDistFeature : public StlFeatureInterface {
public:
    DECLARE_STL_FEATURE_KEY("firsttolastdist");
    DECLARE_STL_FEATURE_TITLE("First to Last Point Distance");
    DECLARE_STL_FEATURE_DESCRIPTION_METHODS;
```

Having setup the identifier key and textual descriptions, the `classCalcValue` member function is declared. This function takes a stroke and returns the feature value for the given stroke in terms of `StlFeatureResultT` which is a floating point number. The last member function `calcValue()` is called by GestureLab whenever the feature needs to be extracted from a stroke during the training or testing stages. `calcValue()` is simply a wrapper for `classCalcValue` and will likely be defined in the feature plugin interface in later versions of GestureLab.

```
    static StlFeatureResultT classCalcValue(const StlStroke & stroke, bool* ok = NULL);

    StlFeatureResultT calcValue(const StlStroke & stroke, bool* ok = NULL) const
    { return classCalcValue(stroke, ok); }
};

#endif // !EXAMPLEFEATURE_H
```

The feature class declaration is now complete.

3.1.2 examplefeature.cpp

Given a declaration of the feature class in `examplefeature.h`, class member functions are implemented in a new file `examplefeature.cpp`. This source file begins with the inclusion of standard C++ libraries for string manipulation and mathematical operations. The `string` library is required by all features to define `classDescription`, whilst the `math` header is included to provide the math functions that are required to calculate the feature value. `examplefeature.h` provides the class declaration developed earlier.

```
#include <string>
#include <cmath>

#include "examplefeature.h"

using namespace std::string;
```

The first member function, `classCalcValue()`, performs the most imporant role for a feature: it computes the value of the feature given an input stroke. In this case, the function returns a value which represents the distance between the first and last points of the stroke. Provided the stroke has at least one point, the Euclidean distance between the first and last points is calculated using the `hypot()` function. The use of `hypot()` explains the inclusion of `<cmath>` earlier. The type `StlFeatureResultT` is a `float`, meaning that feature values are restricted to the range of a standard floating point number.

An `StlStroke` is an STL list of `StlStrokePoint` which contain the public attributes `x`, `y`, `pressure`, and `milliTime`. The `x`, `y` attributes represent the position of the point on the screen and are implemented using floating point values since some tablets scan with resolution that is higher than the display resolution. The `pressure` attribute is used to record the pressure of the pen on the tablet for hardware that supports this function. Pressure information is stored as a floating point value and defaults to zero. Finally, the `milliTime` attribute is an unsigned integer that represents the time (in milliseconds) elapsed since the initial pen down point (which is assigned a time of zero) of the *first* stroke in a gesture.

```
StlFeatureResultT FirstToLastDistFeature::classCalcValue(const StlStroke & stroke,
                                                         bool* ok)
{
    StlFeatureResultT value = 0.0;
    bool resultOk = false;

    if(!stroke.empty())
    {
        const StlStrokePoint & firstPt = stroke.front();
        const StlStrokePoint & lastPt = stroke.back();
        value = hypot(lastPt.x - firstPt.x, lastPt.y - firstPt.y);

        resultOk = true;
    }
```

```
    if(ok != NULL)
        *ok = resultOk;

    return value;
}
```

The remaining member function, `classDescription()`, takes no parameters and simply returns a textual description of the feature. The description may contain basic HTML markup tags and run over several lines, and is shown in GestureLab when features are selected for training.

```
const string & FirstToLastDistFeature::classDescription()
{
    static const string str("Calculates the distance between"
                            "the first and last point.");
    return str;
}
```

`examplefeature.cpp` is now complete and this concludes the implementation of the feature extractor. The feature extractor must however be wrapped and compiled as a plugin for use with Gesture Lab. The plugin class is defined in the following section.

3.1.3 examplefeatureplugin.h

`examplefeatureplugin.h` contains a declaration of the `ExampleFeaturePlugin` class which will provide a recognizer plugin to Gesture Lab by implementing the `StlFeatureFactoryPlugin` interface. This interface requires that `ExampleFeaturePlugin` implement four functions as the following listing shows.

```
#ifndef EXAMPLEFEATUREPLUGIN_H
#define EXAMPLEFEATUREPLUGIN_H

#include "../../GestureRecognition/stlfeaturefactoryplugin.h"

class ExampleFeaturePlugin : public StlFeatureFactoryPlugin {

public:
    std::set<std::string> keys();
    StlFeatureInterface* create(const std::string & key);

    std::string title(const std::string & key);
    std::string description(const std::string & key);
};

#endif  // ! EXAMPLEFEATUREPLUGIN_H
```

The purpose and implementation of these functions is detailed in the following section. Note that the contents of this class declaration is mostly generic; although the class name must be unique for each feature plugin, the general structure of the class is identical for all feature plugins.

3.1.4 examplefeatureplugin.cpp

The implementation of `ExampleFeaturePlugin` member functions is given in `examplefeatureplugin.cpp`. First we must include the header containing the declaration of the plugin class, in addition to the declaration of the feature extractor class. Following this, the `EXPORT_STL_FEATURE_FACTORY_PLUGIN` macro (contained in `stlfeaturefactoryplugin.h`) is used to make the plugin viewable to GestureLab.

```
#include "examplefeatureplugin.h"
#include "examplefeature.h"

using namespace std::string;
using namespace std::set;

EXPORT_STL_FEATURE_FACTORY_PLUGIN(ExampleFeaturePlugin);
```

The member functions of `ExampleFeaturePlugin` are next defined to act as follows: `keys()` returns a set of identifier keys for all features (in this case, one) provided by the plugin; `create()` creates a particular feature extractor object given the identifier key for that feature; `title()` returns a one-line description of the feature given the identifier key for that feature; and `description()` returns a more verbose description of the feature given the identifier key for that feature.

```
set<string> ExampleFeaturePlugin::keys()
{
    set<string> ret;
    ret.insert(FirstToLastDistFeature::classKey());
    return ret;
}

StlFeatureInterface* ExampleFeaturePlugin::create(const string & key)
{
    if(key == FirstToLastDistFeature::classKey())
        return new FirstToLastDistFeature();

    return NULL;
}

string ExampleFeaturePlugin::title(const string & key)
{
    if(key == FirstToLastDistFeature::classKey())
        return FirstToLastDistFeature::classTitle();

    return string();
}

string ExampleFeaturePlugin::description(const string & key)
{
    if (key == FirstToLastDistFeature::classKey())
        return FirstToLastDistFeature::classDescription();

    return string();
}
```

3.1.5 Compiling the Feature Plugin

Source files must be compiled as a `bundle` so that GestureLab can load the new feature as a plugin. The concept of bundles/plugins is specific to Mac OSX and therefore feature extractors are used externally on different platforms using the `librecognizer` library package (see Section 3.4.1). The standard GestureLab build is universal which means that it can run on both PowerPC (PPC) and Intel based Macintosh computers running OSX. To maintain the universal nature of GestureLab, plugins should be compiled as universal bundles and this is achieved using the open source C++ compiler `g++` as follows.

```
g++ -c -pipe -Wall -W -fPIC -arch ppc -arch i386 \
    -isysroot /Developer/SDKs/MacOSX10.4u.sdk -o examplefeature.o \
    examplefeature.cpp

g++ -c -pipe -Wall -W -fPIC -arch ppc -arch i386 \
    -isysroot /Developer/SDKs/MacOSX10.4u.sdk -o examplefeatureplugin.o \
    examplefeatureplugin.cpp

g++ -headerpad_max_install_names -bundle -arch ppc -arch i386 \
    -Wl,-syslibroot,/Developer/SDKs/MacOSX10.4u.sdk \
    -o ExampleFeaturePlugin.bundle examplefeature.o examplefeatureplugin.o
```

The `-arch ppc` and `-arch i386` compilation options ensure that the plugin is built for both PPC and Intel based Macs. The `-isysroot` and `-syslibroot` options indicate that the plugin is being compiled for MacOS 10.4 (Tiger) and requires this version of the operating system or later. Ultimately, the compilation process produces `ExampleFeaturePlugin.bundle`.

3.2 An Example Recognizer Plugin

This tutorial develops a simple GestureLab recognizer plugin. As per feature plugins, recognizer plugins should occupy a separate directory in `digest/src/plugins` and in this case, the source files of the new recognizer plugin are located in `digest/src/plugins/ExampleRecognizer`. **Note:** The following example assumes that `librecognizer` has been unpacked in the directory `digest/src/plugins/ExampleRecognizer/librecognizer` and compiled as a static library according to the instructions given in Section 3.4.1. The plugin uses the `svm_dataset` and `nnsl_mcst_classifier` classes which are provided by `multiclass_svm.h` in the `librecognizer` package; `svm_dataset` is used to build a training dataset, whilst `nnsl_mcst_classifier` is a class which provides functionality to train, load, and save a classifier, and make classification predictions.

3.2.1 svmrecogniser.h

Implementation of the recognizer requires that the recognizer class is declared in a header file we will name `svmrecogniser.h`. This file includes `stlrecogniserinterface.h` for access to the recognizer interface that is to be implemented, as well as including `multiclass_svm.h` to provide the `svm_dataset` and `nnsl_mcst_classifier` classes. Notice that the new recognizer class must inherit `StlRecogniserInterface` (since this is the inferface that all recognizer plugins must implement) and that a `void` constructor will suffice.

```
#ifndef SVMRECOGNISER_H
#define SVMRECOGNISER_H

#include "../../GestureRecognition/stlrecogniserinterface.h"
#include "librecognizer/multiclass_svm.h"

class SvmRecogniser : public StlRecogniserInterface
{
public:
    SvmRecogniser(void);
    ~SvmRecogniser(void);
```

Next, information about the recognizer is set using macros defined in `StlRecogniserInterface`. `DECLARE_STL_RECOGNISER_KEY` sets a identifier string for the recognizer and it is essential that this string is unique to that of all other recognizers. The recognizer key is used by GestureLab internally to identify each recognizer. The `DECLARE_STL_RECOGNISER_TITLE` macro is then used to set a descriptive title for the recognizer. This string appears in the GestureLab training, experiment, and test-pad components when a

recognizer is to be trained or used for testing. The string should therefore provide a concise description of the recognizer, most often, what kind of structure (SVMs, neural networks, etc.) the recognizer employs to classify gestures. Finally, `DECLARE_STL_RECOGNISER_DESCRIPTION` is used to provide methods which set a more verbose textual description of the recognizer.

```
DECLARE_STL_RECOGNISER_KEY("svm_recogniser");
DECLARE_STL_RECOGNISER_TITLE("SVM Recogniser");
DECLARE_STL_RECOGNISER_DESCRIPTION("This is an example recogniser which uses SVMs.");
```

The following segment of code declares the functions required to implement `StlRecogniserInterface`. These functions setup the default recognizer parameters, in addition to providing a means to initialize the training process, examine a particular sample during the training process, and finalize training.

```
const std::map<std::string, std::string> & defaultParams(void) const;
bool initTraining(const std::list<std::string> &,
                  const std::map<std::string, std::string> &);
bool examineSample(const StlFeatureVec &, const std::set<int> &);
bool finaliseTraining(void);
```

`writeModelFile()` and `readModelFile()` functions are necessary so that GestureLab can save a trained recognizer and later load that recognizer for testing. These functions take a single string parameter which specifies the file name of (and optionally, a path to) the trained recognizer. A trained recognizer must therefore be a single file; in the case that the classifier saves trained models over several files (e.g. the multi-class SVM classifiers), it is the responsiblity of the classifier to collect all relevant files into a single archive. In this case, the classifier must also be capable of extracting the files from the archive so that the trained model can be loaded at a later time.

```
bool writeModelFile(const string &);
bool readModelFile(const string &);
```

A gesture consists of one or more ordered strokes and these strokes must undergo a pre-processing stage prior to feature extraction. GestureLab requires that the minimum pre-processing "flattens" the strokes of a gesture into a single stroke i.e., the points of the strokes are concatenated, so that features may be extracted from a single stroke. The `flatten()` virtual function is called prior to examining a stroke during both training and testing stages; the function typically performs pre-processing such as stroke smoothing, jitter removal, point interpolation, etc.

```
StlStroke flatten(const StlStrokeList & strokes);
```

The last function of `StlRecogniserInterface` to implement is `classify()` which, as the name suggests, is called during testing to classify a gesture and return a map of class idenifiers to classfication probabilities. This function takes an `StlFeatureVec` feature vector which has been created by GestureLab automatically, or by an explicit call to `extract_features()` (defined in `common.h` of `librecognizer`) in situations where the recognizer is used external to the GestureLab environment.

```
StlClassProbabilities classify(const StlFeatureVec &);
```

There are only two private data members in this recognizer example: an `svm_dataset` which is used to store the training dataset of feature vectors, and a `nnsl_mcst_classifier` which provides the functionality to train and test an MCST-SVM classifier, in addition to reading/writing the trained classifier from/to file. These classes are defined in `multiclass_svm.h` of `librecognizer`.

```
private:
    svm_dataset training_dataset;
    nnsl_classifier svm_model;
};

#endif  // !SVMRECOGNISER_H
```

3.2.2 svmrecogniser.cpp

Given `svmrecogniser.h`, the accompanying implementation file `svmrecogniser.cpp` is now defined. First, we must include `svmrecogniser.h` to obtain the `SvmRecogniser` class declaration. The constructor and destructor which follow are empty since the recognizer takes no parameters on creation and no data members are dynamically allocated. Note however, this may not necessarily be true for all recognizers.

```
#include "svmrecogniser.h"

using namespace std::list;
using namespace std::string;
using namespace std::map;
using namespace std::set;

SvmRecogniser::SvmRecogniser(void)
{ }

SvmRecogniser::~SvmRecogniser(void)
{ }
```

The `defaultParams()` function is included in the recognizer API so that a set of default parameters may be set for a recognizer and optionally changed a later time using the GestureLab training component. However, the implementation of this software feature is incomplete at the time of writing and therefore `defaultParams()` returns an empty map of parameters to default values. `initTraining()` performs any setup for the recognizer that is required prior to training. The most important parameter to this function is the list of feature keys; the size of this list indicates the number of features which will be used during training. In this example, the `initTraining()` function is used to setup a training dataset with the correct number of training features per sample. This tells the training dataset how many features per sample should be expected when samples are examined during training.

```
const map<string, string> & SvmRecogniser::defaultParams(void) const
{
    static map<string, string> p;
    return p;
}

bool SvmRecogniser::initTraining(const list<string> & featureKeys,
                                 const map<string, string> & params)
{
    training_dataset.set_num_features(featureKeys.size());
    return true;
}
```

Assuming that GestureLab has called `initTraining()` and that the training dataset has been prepared, individual samples can be examined during the training phase. The `examineSample()` function is called each time a training gesture is examined during the training phase. This function takes two parameters: a vector of features extracted from the sample and a set of integers which represent the classes to which the sample belongs. In this example, `examineSample()` adds to the training dataset by inserting a record of a sample's features for each class to which the sample belongs. Since the function is called for each gesture in the training set, a complete training dataset of features is built internally over the course of `examineSample()` calls.

```
bool SvmRecogniser::examineSample(const StlFeatureVec & featureVec,
                                  const set<int> & classes)
{
```

```
    set<int>::const_iterator iter;
    for(iter = classes.begin(); iter != classes.end(); iter++)
        training_dataset.add_sample(*iter, featureVec);

    return true;
}
```

A dataset of training feature vectors will at this stage have been created by GestureLab's calls to `examineSample`. The last step in the training process is therefore to use the training dataset to train the `nnsl_mcst_classifier` data member `svm_model`. In this example, the `finaliseTraining()` function performs the majority of work in the training process: the training dataset is first scaled so that each feature is in the range $[-1, 1]$ and the scaled dataset is then used passed to the MCST-SVM classifier for training. Using this type of classifier, the length of the training process is generally in the order of minutes to hours for datasets of 3 or more classes comprising 50 gestures each. When `finaliseTraining` returns, the recognizer is trained for use and may be written to file for later reference.

```
bool SvmRecogniser::finaliseTraining(void)
{
    training_dataset.scale();
    svm_model.train(training_dataset);
    return true;
}
```

The `writeModelFile()` and `readModelFile()` member functions provide recognizer persistence to Gesture Lab. The `writeModelFile()` function is called when a recognizer completes training and, in this case, the function simply acts as a wrapper for the `save()` function implemented by the `nnsl_mcst_classifier` class. `writeModelFile()` is called when GestureLab starts and is used to load all previously trained recognizers that are recorded in the GestureLab database. Here, `readModelFile()` is a wrapper function which calls the `load()` function implemented by the `nnsl_mcst_classifier` class and returns the success or failure of this operation. The `save()` and `load()` functions take a single string parameter which specifies the path to the trained recognizer file and the name of the recognizer model file.

```
bool SvmRecogniser::writeModelFile(const string & filePath)
{
    return svm_model.save(filePath);
}

bool SvmRecogniser::readModelFile(const string & filePath)
{
    bool result = svm_model.load(filePath);
    return result;
}
```

The example `flatten()` function that follows creates a flattened stroke by concatenating the points which comprise each stroke in the stroke list (passed as a parameter). The flattened stroke is formed using the generic `copy()` and `back_inserter()` functions provided by the standard C++ library. Once created, the flattened stroke is returned with no additional pre-processing.

```
StlStroke SvmRecogniser::flatten(const StlStrokeList & strokes)
{
    StlStroke flattened_stroke;
    StlStrokeList::const_iterator iter;

    for(iter = strokes.begin(); iter != strokes.end(); iter++)
        copy((*iter).begin(), (*iter).end(), back_inserter(flattened_stroke));
```

```
    return flattened_stroke;
}
```

`classify()` is the final function of `StlRecogniserInterface` to implement. In this case, `classify()` acts simply as a wrapper for the `classify_with_leaf()` function provided by the class `nnsl_mcst_classifier`. `classify_with_leaf()` is `nnsl_mcst_classifier` and in this case the classifier takes three parameters: a vector of features which have been extracted from the gesture; an indicator of whether the features have been scaled; and a map to store classification probabilities for each possible class. The second parameter is required to calculate confusion distributions in each leaf of the classifier tree, however this use of `classify_with_leaf()` is "internal" to the classifier and should not be of concern to a developer. In testing situations, the second parameter should always indicate that features have not been scaled.

```
StlClassProbabilities SvmRecogniser::classify(const StlFeatureVec & featureVec)
{
    StlClassProbabilities classification_probs;
    svm_model.classify_with_leaf(featureVec, nnsl_classifier::unscaled,
                                 classification_probs);

    return classification_probs;
}
```

The implementation of the `SvmRecogniser` class is now complete. Recognizers must be wrapped and compiled as a plugin for use with GestureLab and this plugin is developed in the following sections.

3.2.3 svmrecogniserplugin.h

`svmrecogniserplugin.h` contains a declaration of the `SvmRecogniserPlugin` class which will provide a recognizer plugin to GestureLab by implementing the `StlRecogniserFactoryPlugin` interface. This interface requires that `SvmRecogniserPlugin` implements four functions as the following listing shows.

```
#ifndef SVMRECOGNISERPLUGIN_H
#define SVMRECOGNISERPLUGIN_H

#include "../../GestureRecognition/stlrecogniserfactoryplugin.h"

class SvmRecogniserPlugin : public StlRecogniserFactoryPlugin
{
public:
    std::set<std::string> keys();
    StlRecogniserInterface* create(const std::string & key);

    std::string title(const std::string & key);
    std::string description(const std::string & key);
};


#endif  // !SVMRECOGNISERPLUGIN_H
```

The purpose and implementation of these functions is detailed in the following section. Note that the declaration of the recognizer plugin is mostly generic; although the class name may change, the general structure of the class is the same for all recognizer plugins.

3.2.4 svmrecogniserplugin.cpp

`svmrecogniserplugin.cpp` provides the implementation of the member functions declared in `svmrecogniserplugin.h`. First we must include the header containing the declaration of the plugin class, in addition to the declaration of the recognizer class. Following this, the `EXPORT_STL_RECOGNISER_FACTORY_PLUGIN` macro (provided by `stlrecogniserfactoryplugin.h`) is used to make the plugin viewable to GestureLab.

```
#include "svmrecognizer.h"
#include "svmrecogniserplugin.h"

using namespace std::string;
using namespace std::set;

EXPORT_STL_RECOGNISER_FACTORY_PLUGIN(SvmRecogniserPlugin);
```

The member functions of `SvmRecogniserPlugin` are next defined to act as follows: `keys()` returns a set of identifier keys for all recognizers (in this case, one) provided by the plugin; `create()` creates a particular recognizer object given the identifier key for that recognizer type; `title()` returns a one-line description of the recognizer given the identifier key for that recognizer type; and `description()` which returns a more verbose description of the recognizer given the identifier key for that recognizer type.

```
set<string> SvmRecogniserPlugin::keys()
{
    set<string> keyset;
    keyset.insert(SvmRecogniser::classKey());
    return keyset;
}

StlRecogniserInterface* SvmRecogniserPlugin::create(const std::string & key)
{
    if(key == SvmRecogniser::classKey())
        return new SvmRecogniser();

    return NULL;
}

string SvmRecogniserPlugin::title(const std::string & key)
{
    if(key == SvmRecogniser::classKey())
        return nnsl_mcst_recogniser::classTitle();

    return string();
}

string SvmRecogniserPlugin::description(const std::string & key)
{
    if (key == SvmRecogniser::classKey())
        return SvmRecogniser::classDescription();

    return string();
}
```

3.2.5 Compiling the Recognizer Plugin

Like the feature plugin, recognizer source files must be compiled as a `bundle` for use with GestureLab. To maintain the universal nature of GestureLab, recognizer plugins should be compiled as universal bundles and this is achieved using the open source C++ compiler `g++` as follows.

```
g++ -c -pipe -Wall -W -fPIC -arch ppc -arch i386 \
    -isysroot /Developer/SDKs/MacOSX10.4u.sdk -o svmrecogniser.o \
    svmrecogniser.cpp

g++ -c -pipe -Wall -W -fPIC -arch ppc -arch i386 \
    -isysroot /Developer/SDKs/MacOSX10.4u.sdk -o svmrecogniserplugin.o \
    svmrecogniserplugin.cpp

g++ -headerpad_max_install_names -bundle -arch ppc -arch i386 \
    -Wl,-syslibroot,/Developer/SDKs/MacOSX10.4u.sdk -o SvmRecogniser.bundle \
    svmrecogniser.o svmrecogniserplugin.o \
    librecognizer/librecognizer.a librecognizer/libtar/lib/libtar.a
```

The `-arch ppc` and `-arch i386` compilation options ensure that the plugin is built for both PPC and Intel based Macs. The `-isysroot` and `-syslibroot` options indicate that the plugin is being compiled for MacOS 10.4 (Tiger) or later. The `librecognizer.a` library (see Section 3.4.1) must be linked since `SvmRecogniser` contains `training_dataset` and `nnsl_mcst_classifier` data members which are provided by `librecognize`. The use of `librecognizer` in turn requires that `libtar` is also linked. A recognizer plugin that does not make use of `librecognizer` classes and functions will not require linking against these static libraries.

Ultimately, the compilation process produces `SvmRecogniser.bundle`.

3.3 Plugin Installation

Viewed from a command shell, GestureLab appears as a directory `GestureLab.app` with subdirectories including `PlugIns` and `PlugIns Disabled`. `SvmRecogniser.bundle` and `ExampleFeaturePlugin.bundle` must be copied into `GestureLab.app/Contents/PlugIns` before GestureLab can use the feature extractors and recognizers provided by these plugins. The contents of the application can also be accessed via a GUI by right-clicking the GestureLab program icon and selecting "Show Package Contents" from the context menu. Plugins can be disabled by moving bundles from `GestureLab.app/Contents/PlugIns` to `GestureLab.app/Contents/PlugIns Disabled`.

Alternatively, plugins can be activated and deactivated via the GUI by right-clicking the GestureLab program icon and selecting "Get Info". Each installed plugin is listed alongside a check box; unchecking/checking the box will disable/enable the corresponding plugin. This is a graphical front-end to the process of moving bundles between `GestureLab.app/Contents/PlugIns` and `GestureLab.app/Contents/PlugIns Disabled`.

3.4 Recognizer Integration

3.4.1 Overview of librecognizer

GestureLab uses Mac OSX bundles and the associated software plugin framework to provide support for generic recognizers and feature extractors. Whilst this is an effective way to implement plugins, the use of bundles ties plugins to the OSX operating system. Consequently, a separate `librecognizer` package of C++/Java source files has been created. The `librecognizer` package contains all source code required to build a library of the default GestureLab features and recognizers in a platform independent manner. Once compiled on the target platforum, this library can be used to load trained recognizers from file and classify gestures using stroke data captured by the particular front-end application. `librecognizer` currently provides the DAG-SVM [3] recognizer, four varieties of MCST-SVM recognizer [2], in addition

to a set of feature extractors which implement the 13 features described in [4]. The software package is available for download at `http://www.csse.monash.edu.au/~berndm/gesture_lab/GL/`.

Two Makefiles are included in the `librecognizer` distribution: `Makefile.Mac` and `Makefile.Other`. `Makefile.Mac` compiles `librecognizer` as a universal (PPC and Intel) static library for Mac OSX; this form of compilation is appropriate when the library will be used to create new recognizer and/or feature plugins for GestureLab, or integrate the recognizers in an OSX application. `Makefile.Other` is used to compile the library so that it can be used to integrate GestureLab recognizers with applications that are developed on non-OSX platforms such as Linux. A Java wrapper class `recognizerProxy` is included to provide a means to integrate an `nnsl_mcst_recogniser` recogniser with Java front-ends; the `nnsl_mcst_recogniser` class was chosen for wrapping because it has demonstrated high performance for many-way gesture classification problems.

Additional GestureLab plugins should be developed using a self-contained library in the way that `librecognizer` has been used to develop the default recognizer and feature plugins. This approach means that the library can be used to integrate quickly the recognizers and feature extractors into external applications by means of a simple linking process. Since front-end applications may run on a variety of platforms, feature extractor and recognizer source code should be kept platform independant.

3.4.2 C/C++ Integration

Integration of the default GestureLab recognizers in a C/C++ application is straight-forward since these recognizers are implemented using C++. This tutorial develops a C++ program in `example.cpp` which demonstrates how trained recognizers trained in GestureLab may be loaded externally and used to classify gestures.

The program begins by including the relevant header files. `<iostream>` is included primarily so that the results of classification can be reported, but is also necessary to report error messages.
`nnsl_mcst_recogniser.h` is included in the `librecognizer` package and provides a declaration of the recognizer which will be used to classify gestures. This class must match the type of recognizer trained in GestureLab and it is the developer's responsibility to ensure that this is the case. `stlrecognitioncommon.h` provides types which represent gesture strokes, points, and a map of target classes to classification probabilities. These types will be used later in the example. `common.h` is part of the `librecognizer` distribution and is included so that the `extract_features()` function may be used. `extract_features()` is required to extract features from a gesture when a trained recognizer is used in a standalone setting (separate from GestureLab) and at present, the features extracted and the order of these features match those extracted by the "Standard Features" GestureLab plugin. It is the developer's responsibility to ensure that the feature set used to train a recognizer in GestureLab and the feature set returned by `extract_features()` match. This may require modification of the `extract_features()` function, however such modification is trivial.

```
#include <iostream>

#include "librecognizer/common.h"
#include "librecognizer/nnsl_mcst_recogniser.h"
#include "librecognizer/stlrecognitioncommon.h"

using namespace std;

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

The next section of code begins the main program block and creates a new recognizer object. A simple check is performed to ensure that the recognizer was allocated correctly and the program halts if this is not the case.

```
int main()
{
    nnsl_mcst_recogniser *recognizer = new nnsl_mcst_recogniser();

    if(!recognizer)
    {
        cout << "Error: could not allocate a new recognizer!" << endl;
        exit(EXIT_FAILURE);
    }
```

Given a recognizer object, the trained recognizer can now be loaded from the `.model` file produced by GestureLab. The following code segment attempts to load the trained recognizer `nnsl_mcst_recognizer12.model` and halts if this is unsuccessful (e.g. the model file is not found).

```
    if(!recognizer->readModelFile("nnsl_mcst_recognizer12.model"))
    {
        cout << "Error: could not load the trained recognizer from file!" << endl;
        exit(EXIT_FAILURE);
    }
```

At this stage the program has loaded a trained recognizer from file. The next step involves creating a stroke manually so that it may be classified. A "real-world" integration of the GestureLab recognizers would not create strokes in this contrived way; instead, strokes would be created using pen-based input from a graphics tablet. Nevertheless, the following code demonstrates how strokes are created from a series of points.

Storage for two "drawn" strokes is first allocated in the form of `StlStroke` objects. The "drawn" stroke comprises three `StlStrokePoint` objects which are created using a set of parameters representing `(x, y, pressure, time)` point attributes. Points are pushed onto the back of each `StlStroke` in series to maintain correct ordering. The two strokes created here form a gesture of two vertical lines running from coordinates $(50, 40)$ to $(50, 60)$ and $(50, 70)$ to $(50, 90)$ respectively. Each point occurs 10 milliseconds after the previous point.

```
    StlStroke firstStroke, secondStroke;

    StlStrokePoint point1(50, 40, 0,  0);
    StlStrokePoint point2(50, 50, 0, 10);
    StlStrokePoint point3(50, 60, 0, 20);
    firstStroke.push_back(point1);
    firstStroke.push_back(point2);
    firstStroke.push_back(point3);

    StlStrokePoint point4(50, 70, 0, 30);
    StlStrokePoint point5(50, 80, 0, 40);
    StlStrokePoint point6(50, 90, 0, 50);
    secondStroke.push_back(point4);
    secondStroke.push_back(point5);
    secondStroke.push_back(point6);
```

The program next creates an `StlStrokeList` which contains the two strokes created above. Each `StlStroke` appended to the `StlStrokeList` in the order that the strokes were "drawn"; correct ordering must be maintained at all times to ensure that the gesture is recognized correctly.

```
    StlStrokeList strokes;
    strokes.push_back(firstStroke);
    strokes.push_back(secondStroke);
```

The `StlStrokeList` of strokes can now be given as input to the recognizer's `flatten()` function. This function applies pre-processing to the strokes and returns a single "flattened" representation of the `StlStrokeList`.

```
    StlStroke processedStroke = recognizer->flatten(strokes);
```

A set of features is extracted from the pre-processed (or "flattened") stroke using the `extract_features()` function provided by `librecognizer`. An `StlFeatureVec` is simply an STL `Vector` of doubles.

```
    StlFeatureVec features = extract_features(processedStroke);
```

The feature set can now be used to classify the gesture probabilistically using the recognizer's `classify()` function. This function returns a `StlClassProbabilities` map which is an STL map of integer class identifiers to classification predictions stored with `double` precision.

```
    StlClassProbabilities classificationMap = recognizer->classify(features);
```

The following section of code outputs the probability of the stroke belonging to each of the classes for which the recognizer was trained. That is, the code outputs each key in `StlClassProbabilities` map and the corresponding prediction probability.

```
    StlClassProbabilities::const_iterator iter;

    for(iter = classificationMap.begin(); iter != classificationMap.end(); iter++)
        cout << "Class: " << iter->first << "\tProbability:" << iter->second << endl;
```

Finally, the program de-allocates the recognizer and returns an indication of successful completion.

```
    delete recognizer;

    return EXIT_SUCCESS;
}
```

The program is now complete. Assuming the `librecognizer` package in a sub-directory of the same name, compilation and execution of the program is achieved using the following commands.

```
    g++ -o example -DFOR_EXTERNAL_USE example.cpp \
    librecognizer/librecognizer.a librecognizer/libtar/lib/libtar.a

./example
```

The `FOR_EXTERNAL_USE` symbol is defined to obtain access to the `extract_features()` function prototype in `common.h`. This function is not required when the recognizer is used in the GestureLab environment since GestureLab manages feature extraction internally.

3.4.3 Java Integration

Java Native Interface (JNI) is used to integrate into a Java application the recognizers created by GestureLab. This is achieved using a proxy (or "wrapper") Java class which provides functions that are mapped to the most important recognizer routines such as loading a recognizer and classifying a sample. A wrapper function to train a new recognizer is not provided since training is one of GestureLab's roles.

The nature of JNI means that the proxy class is tied to a specific class of Gesture Lab recognizer; the proxy implemented for `librecognizer` targets the Minimal Cost Spanning Tree (MCST) Nearest Neighbour Single Linkage (NNSL) recognizer (identified as NNSL-MCST). This class of recognizer was selected on the basis of high performance achieved for large classification problems, however modification

of the proxy to target a different recognizer class is straight-forward. The recognizer proxy is defined in `recognizerProxy.java` of the `librecognizer` package.

This tutorial develops a Java program `example.java` which demonstrates how trained recognizers may be loaded and used to classify gestures in a Java environment. Assuming the `librecognizer` package has been unpacked in the current working directory, the program begins by importing the `recognizerProxy` class.

```
import librecognizer.recognizerProxy;
```

The next section of code begins the class definition and the `main()` function. `main()` first creates a new `recognizerProxy` object and checks that the object was created successfully; if this is not the case, the program simply reports an error and halts.

```
public class example
{
    public static void main(String args[])
    {
        recognizerProxy recognizer = new recognizerProxy();

        if(recognizer == null)
        {
            System.out.println("Error: could not allocate a new recognizer proxy!");
            System.exit(1);
        }
```

At this stage, the `recognizerProxy` object can be used to load a trained recognizer from a `.model` file. Here, the example loads `nnsl_mcst_recognizer12.model` which is the result of training a new recognizer using GestureLab. Note that the prefix of the file name indicates the type of recognizer which has been trained (NNSL-MCST).

```
        if(!recognizer.readModelFile("nnsl_mcst_recognizer12.model"))
        {
            System.out.println("Error: could not load the trained " +
                                "recognizer from file!");
            System.exit(1);
        }
```

The following section of code creates two strokes in the serialized form required by `recognizerProxy`. The proxy represents strokes using 1-dimensional array of serialized point data, where each point is an ordered tuple of `(x, y, time)` floats. Strokes are delimited in the array using a reserved value `-1`. The following example creates two strokes which each comprise three points each; these strokes are identical to those used in the C++ example of Section 3.4.2.

```
        float serializedStroke[] = {50, 40,  0,
                                    50, 50, 10,
                                    50, 60, 20,
                                    -1,
                                    50, 70, 30,
                                    50, 80, 40,
                                    50, 90, 50,
                                    -1};
```

The program can now classify the gesture to obtain probabilistic predictions. This is achieved by a call to `classifySampleProb()` implemented by `recognizerProxy`. `classifySampleProb()` takes the serialized stroke as a single parameter and returns an array that represents a serialized map of class identifiers to prediction probabilities. The contents of the array returned are arranged so that class identifiers are always followed by the probability of the gesture belonging to that class.

```
        float[] serializedProbabilityMap =
                            recognizer.classifySampleProb(serializedStroke);
```

Note that no explicit function calls are made to pre-process the gesture data or extract features from the pre-processed strokes. These routines are performed by `classifySampleProb()` with the aim of reducing the number of functions that need to be wrapped using JNI.

At this stage, a recognizer has been loaded from file and used to classify a gesture. The remaining code outputs the probability of the gesture belonging to each of the classes for which the recognizer was trained. Each iteration of the loop outputs a particular class identifer and its associated prediction probability.

```
        for(int i = 0; i < serializedProbabilityMap.length; i += 2)
            System.out.println("Class: " + serializedProbabilityMap[i] +
                               "\tProbability: " + serializedProbabilityMap[i+1]);
    }
}
```

The example program is now complete. The `librecognizer` directory must be added to the Java class path so that the `recognizerProxy` class can be found by the Java Runtime Environment. Compilation and execution of the program is therefore achieved using the following commands.

```
javac example.java
java -classpath .:librecognizer example
```

References

[1] A. R. Jansen, K. Marriott, and B. Meyer. Cider: A component-based toolkit for creating smart diagram environments. In *Proceedings of the International Conference on Distributed and Multimedia Systems*, 2003.

[2] A. C. Lorena and A. C. P. L. F. de Carvalho. Minimum spanning trees in hierarchical multiclass support vector machines generation. *Innovations in Applied Artificial Intelligence*, 3533:422–431, 2005.

[3] J. C. Platt, N. Cristinini, and J. Shawe-Taylor. Large margin DAGs for multiclass classification. *Advances in Neural Information Processing Systems*, 12:547–553, 2000.

[4] D. Rubine. Specifying gestures by example. *Computer Graphics*, 25(4):329–337, 1991.