The Gamera framework for building custom recognition systems

Michael Droettboom **Karl MacMillan** **Ichiro Fujinaga**
Digital Knowledge Center, Sheridan Libraries
The Johns Hopkins University
3400 N. Charles St., Baltimore, MD 21218
mdboom@jhu.edu, karlmac@jhu.edu, ich@music.mcgill.ca

Abstract

This paper describes the Gamera framework for building custom document recognition systems. This open-source system is designed to support the test-and-refine development cycle: an important style for developing recognition systems that work with difficult historical documents, since the solutions are often non-obvious. This paper explains the overall architecture of the system, in addition to detailed information on recent research subprojects and their performance on real-world data.

1 Introduction

Gamera is a framework for the creation of structured document analysis applications by domain experts. Domain experts are individuals who have a strong knowledge of the documents in a collection, but may not have a formal technical background. The goal is to create a tool that leverages their knowledge of the target documents to create custom applications rather than attempting to meet diverse requirements with a monolithic application.

We intend to use Gamera to develop applications for a number of diverse collections, including medieval manuscripts, lute tablature, Greek text, and handwritten text. There are many collections that would benefit from this technology and would represent a diverse range of document analysis challenges [1].

This paper gives an overview of the architecture and design principles of Gamera. In addition, recent additions to Gamera are discussed in more detail, with an analysis of their performance on real-world data.

2 Architecture overview

Developing recognition systems for difficult historical documents requires experimentation since the solution is often non-obvious. Therefore, Gamera's primary goal is to support an efficient test-and-refine development cycle. Virtually every implementation detail is driven by this goal. For instance, Python [2] was chosen as the core language because of its introspection capabilities, dynamic typing and ease of use. It has been used as a first programming language with considerable success [3]. `C++` is used to write plugins where runtime performance is a priority, but even in that case, the Gamera plugin system (Section 2.3.2) is designed to make writing extensions as easy as possible. Gamera includes a full-fledged graphical user interface that provides a number of shortcuts for training, as well as inspection of the results of algorithms at every step. By improving the ease of experimentation, we hope to put the power to develop recognition systems with those who understand the documents best. We expect at least two kinds of developers to work with the system: those with a technical background adding algorithms to the system, and those working on the higher-level aggregation of those pieces. It is important to note this distinction, since those groups represent different skill sets and requirements.

In addition to its support of test-and-refine development, Gamera also has several other advantages that are important to large-scale digitization projects in general. These are:

- Open source code and standards-compliance so that the software can interact well with other parts of a digitization framework
- Platform independence, running on a variety of operating systems including Linux (Figures 2–5) and Microsoft Windows (Figures 8–9)
- A workflow system to combine high-level tasks
- Batch processing
- A unit-testing framework to ensure correctness and avoid regression
- Rich user interface components for development and classifier training

- Recognition confidence output so that collection managers can easily target documents that need correction or different recognition strategies

2.1 Tasks

Gamera has a modular plugin architecture. These modules typically perform one of five document recognition tasks:

1. Pre-processing
2. Document segmentation and analysis
3. Symbol segmentation and classification
4. Syntactical or structural analysis
5. Output

Each of these tasks can be arbitrarily complex, involve multiple strategies or modules, or be removed entirely depending on the specific recognition problem at hand. The actual steps that make up a complete recognition system are completely controlled by the user.

2.1.1 Pre-processing

Preprocessing involves standard image-processing operations such as noise removal, blurring, deskewing, contrast adjustment, sharpening, binarization, and morphology. Close attention to and refinement of these steps is particularly important when working with degraded historical documents.

2.1.2 Document segmentation and analysis

Before the symbols of a document can be classified, an analysis of the overall structure of the document may be necessary. The purpose of this step is to analyze the structure of the document, segment it into sections, and perhaps identify and remove elements. For example, in the case of music recognition, it is necessary to identify and remove the staff lines in order to be able to properly separate the individual symbols. Similarly, text documents may require the removal of figures.

2.1.3 Symbol segmentation and classification

The segmentation, feature extraction, and classification of symbols is the core of the Gamera system. The system allows different classifiers to be plugged-in. These classifiers come in two flavors: "interactive" classifiers, where examples can be added and the results tested immediately, and "non-interactive" classifiers, that are highly optimized but static. Gamera has a generic and flexible XML-based file format (Section 2.1.5) to store classifier data, but for efficiency, the "non-interactive" classifiers can also define their own format containing Pre-parsed or pre-optimized data. At present, we have an implementation of the k-nearest neighbor (k-NN) algorithm [4] whose weights are optimized using a genetic algorithm (GA) [5]. We have tested the extensibility of our classifier framework by porting a simple back-propagating neural network library to Gamera [6]. We also plan to examine what modifications would be necessary to support stateful classifiers, such as hidden Markov models.

2.1.4 Syntactical or structural analysis

This process reconstructs a document into a semantic representation of the individual symbols. Examples of this include combining stems, flags, and noteheads into musical objects, or grouping words and numbers into lines, paragraphs, columns etc. Obviously, this process is entirely dependent on the type of document being processed and is a likely place for large customizations by knowledgeable users.

Section 3.1 describes a pattern matching engine designed to help with many of these analysis problems.

2.1.5 Output

Gamera stores groups of glyphs in an XML-compliant format (Figure 1). This makes it very easy to save, load, and merge sets of training data. Since a run-length encoded copy of the glyph is included, it is easy to load the original images and inspect, edit or generate new features from them.

Output of data that is specific to a particular type of document, i.e. post-structural interpretation, is deliberately left open-ended, since different domains will have different requirements. For example, the GUIDO [7] file format is used for symbolic music representation by our Gamera-based music recognition application.

2.2 Graphical user interface

Since document recognition is an inherently visual problem, a graphical user interface (GUI) is included to allow the application developer to experiment with different recognition strategies. At the core of the interface is the console window (Figure 2) which allows the programmer to run code interactively and control the system either by typing commands or using menus. All commands are recorded in a history, which can later be used for building automatic batch scripts. The interface also includes a simple image viewer, image analysis tools (Figure 3), and a training interface for the learning classifiers (Figure

Figure 1: An example glyph from the Gamera XML file format.

```
<?xml version="1.0" encoding="utf-8"?>
<gamera-database version="2.0">
  <glyphs>
    <glyph uly="798" ulx="784" nrows="15" ncols="12">
      <ids state="MANUAL">
        <id name="lower.c" confidence="1.000000"/>
      </ids>
      <!-- Run-length encoded binary image (white first) -->
      <data>
        5 6 4 9 2 4 2 4 2 4 3 3 1 4 6 5 8 4 9 3 8 4 9 4 8 5 7 5 3 3 3 9 3 9 6
        4 2 0
      </data>
      <features scaling="1.0">
        <feature name="area">
          180.0
        </feature>
        <feature name="aspect_ratio">
          0.8
        </feature>
        <feature name="compactness">
          0.584269662921
        </feature>
        <feature name="moments">
          0.219907407407 0.228888888889 0.0697385116598 0.126611111111
          0.0505606995885 0.0203254388586 0.0177776861746 0.00727370913662
          0.0488995911061
        </feature>
        ...
      </features>
    </glyph>
    ...
  </glyphs>
</gamera_database>
```

4). The training interface allows the user to create databases of labeled glyphs, including through the merging and splitting connected components.

The interface can easily be extended to include new elements as modules are added to Gamera. The entire GUI is written in Python, using the wxPython toolkit [8].

2.3 Implementation details

2.3.1 Image classes

The storage and manipulation of images is one of the most important aspects of Gamera. Gamera must provide not only general-purpose image manipulation functions, but also infrastructure to support the symbol segmentation and analysis. The features of the `Image` classes in Gamera include:

Polymorphic image types. Gamera images can be in stored using a number of different pixel types, including color (24-bit RGB), greyscale (8- and 16-bit), floating point (32-bit) and bi-level images, though new images types can be added as desired.

Consistent programming interface. The interface to all types of images (in both `C++` and Python) is the same. While some methods are not available for all image types (e.g. thresholding a bi-level image would not make sense), in many cases image types are interchangeable, meaning types can be changed at different points in the development process.

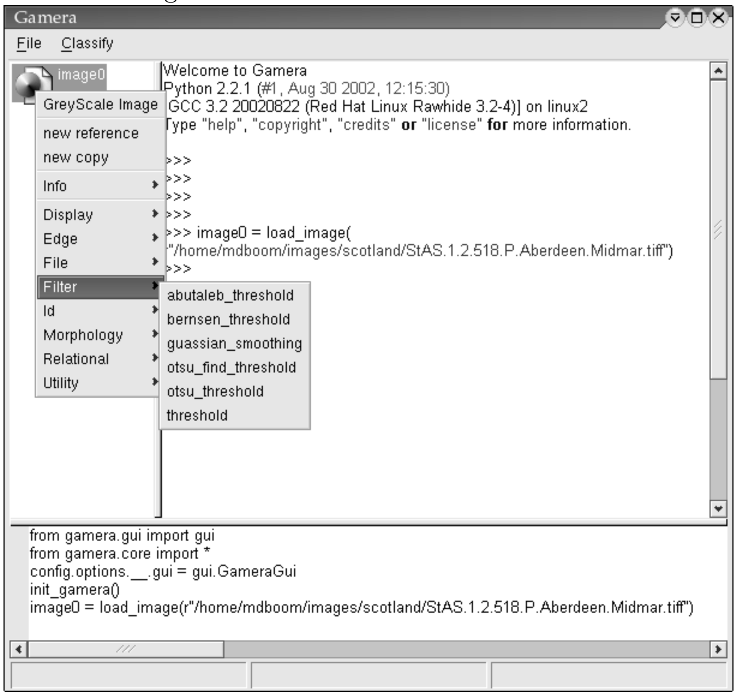
Use of existing code libraries. The image classes have been designed especially to make transferring code from other `C/C++` image processing libraries as easy as possible. For example, many algorithms in the VIGRA library [9] can be used in Gamera without modification. Using `C` libraries, such as XITE [10], often requires only a few minor modifications of the code. This ability has reduced our development time considerably.

Portions of images. The image classes allow for the flexible and efficient representation of portions of images, including non-rectangular regions, without resorting to memory copying. The bi-level images actually store 16-bits-per-pixel, so that labeling information can be stored to define connected components. This uses a considerably smaller memory footprint than using separate data for each connected component.

2.3.2 The plugin system

Writing wrapper code to call `C/C++` from Python is a time-consuming, error-prone and repetitive task. A number of general-purpose tools exist to help automate this process, including SWIG (`http://www.swig.org`) and Boost Python (`http://www.boost.org`). In fact, an earlier version of Gamera used Boost Python, but the additional function-call overhead for our highly polymorphic image types lead to poor performance of that system as a whole. We have since developed our own

Figure 2: The Gamera console window.



```
Welcome to Gamera
Python 2.2.1 (#1, Aug 30 2002, 12:15:30)
[GCC 3.2 20020822 (Red Hat Linux Rawhide 3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>>
>>>
>>>
>>>
>>> image0 = load_image(
r"/home/mdboom/images/scotland/StAS.1.2.518.P.Aberdeen.Midmar.tiff")
>>>
```

```
from gamera.gui import gui
from gamera.core import *
config.options.__.gui = gui.GameraGui
init_gamera()
image0 = load_image(r"/home/mdboom/images/scotland/StAS.1.2.518.P.Aberdeen.Midmar.tiff")
```

wrapper-generating mechanism specific to Gamera and its classes. This allowed us to provide optimizations and conveniences to the programmer that would not be possible with a more general approach.

To add a plugin function to Gamera, a programmer writes metadata about the function (in Python) and a single function to perform an image-processing task (in C++). Plugin functions are grouped into standard Python modules, which are collections of related classes and functions.

As an example, the metadata for the Morphology module (morphology.py) is listed below.

```
class erode_dilate(PluginFunction):
  self_type = ImageType([ONEBIT, GREYSCALE, FLOAT])
  args = Args([
    Int('times',
        range=(0, 10), default=1),
    Choice('direction',
           ['dilate', 'erode']),
    Choice('window_shape',
           ['rectangular', 'octagonal'])
    ])
...

class MorphologyModule(PluginModule):
  cpp_headers = ["morphology.hpp"]
  cpp_namespaces = ["Gamera"]
  category = "Morphology"
  functions = [erode_dilate, erode, dilate,
               rank, mean]
  author = "Michael␣Droettboom␣and␣Karl␣MacMillan"
  url = "http://gamera.dkc.jhu.edu/"
```

Each plugin function has a class that inherits from PluginFunction. There are a number of (static) members of this class that can be used to customize the function. self_type lists the types of images that the function can be performed on. This information is important, since we must map the polymorphic method calls made in the Python environment to a particular compiled instance of the function in C++. The args member gives details about the function's arguments. This detail goes beyond what could be obtained automatically from a C++ function declaration to include elements such as ranges and named enumerations. This information is used both to automatically generate the function wrapper code and to generate dialog boxes within the GUI (Figure 5). At the module level, there is a single class to describe the entire module that inherits from PluginModule. This is used to specify which C++ headers or libraries need to be included, authorship information, and other miscellaneous hints to the C++ compiler. The plugin functions themselves are just free C++ functions. The declaration of the function for erode_dilate (in morphology.hpp) is given below. (The details of the algorithm itself are not important to understanding the plugin system.)

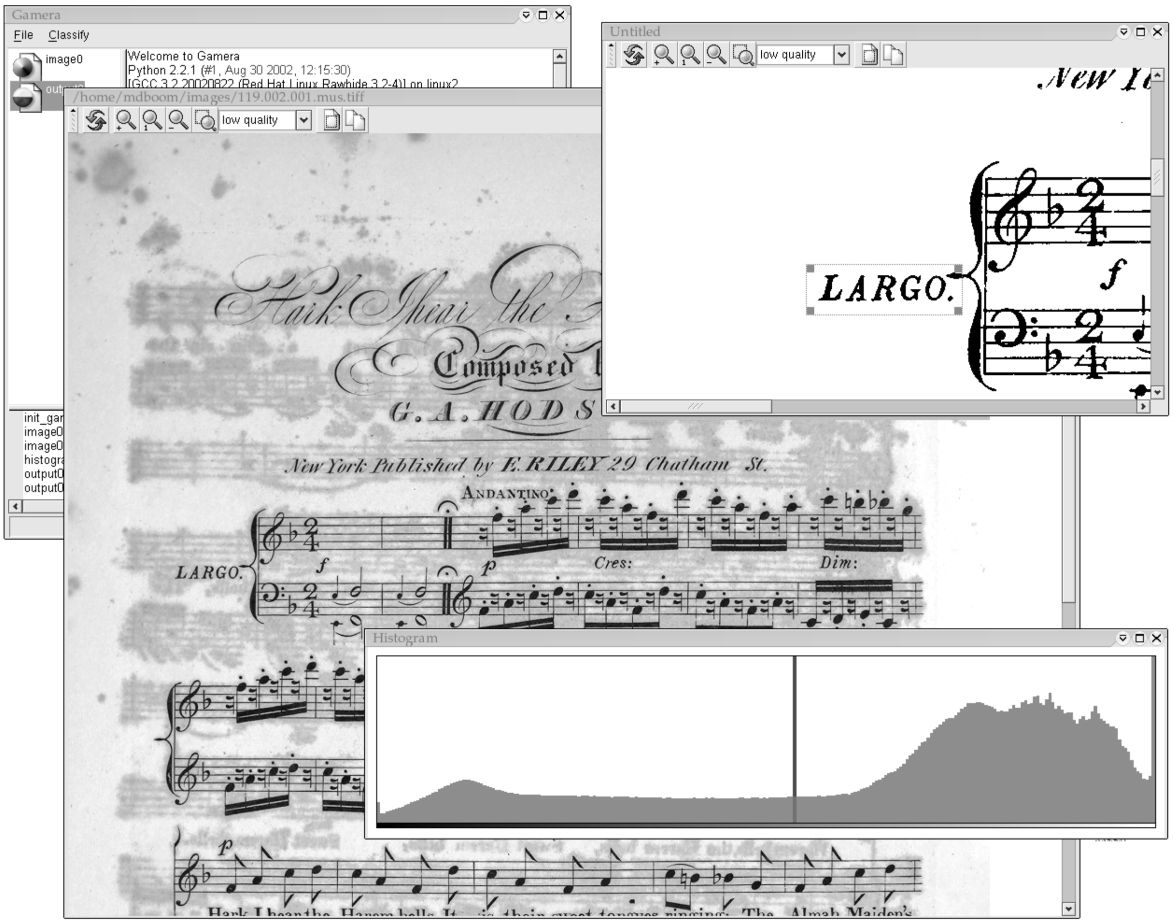
```
template<class T>
void erode_dilate(T &m, unsigned int times,
                  int direction, int window_shape) {

  ...

}
```

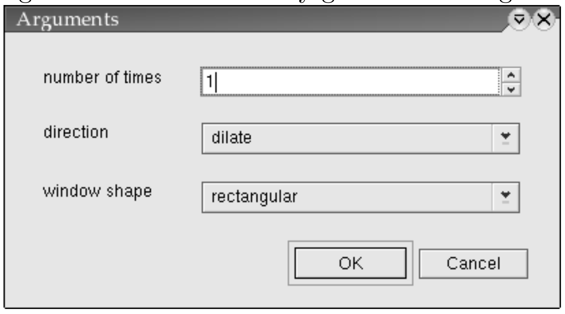
The Gamera build system uses the above metadata to generate a wrapper for the function. Since the function is templatized, a different instance

Figure 3: The Gamera image viewer.



```
Welcome to Gamera
Python 2.2.1 (#1, Aug 30 2002, 12:15:30)
[GCC 3.2 20020822 (Red Hat Linux Rawhide 3.2-4)] on linux2
```

Figure 5: An automatically generated dialog box.



number of times	1
direction	dilate
window shape	rectangular

of the function is compiled for each of the image types it supports. The plugin functions are added to the Gamera `Image` class at runtime. This makes it possible to use standard dot syntax to call these methods. For example, the plugin function `erode_dilate` could be performed on image `img0` with the statement `img0.erode_dilate(0, 0, 0)` (instead of `erode_dilate(img0, 0, 0, 0)`).

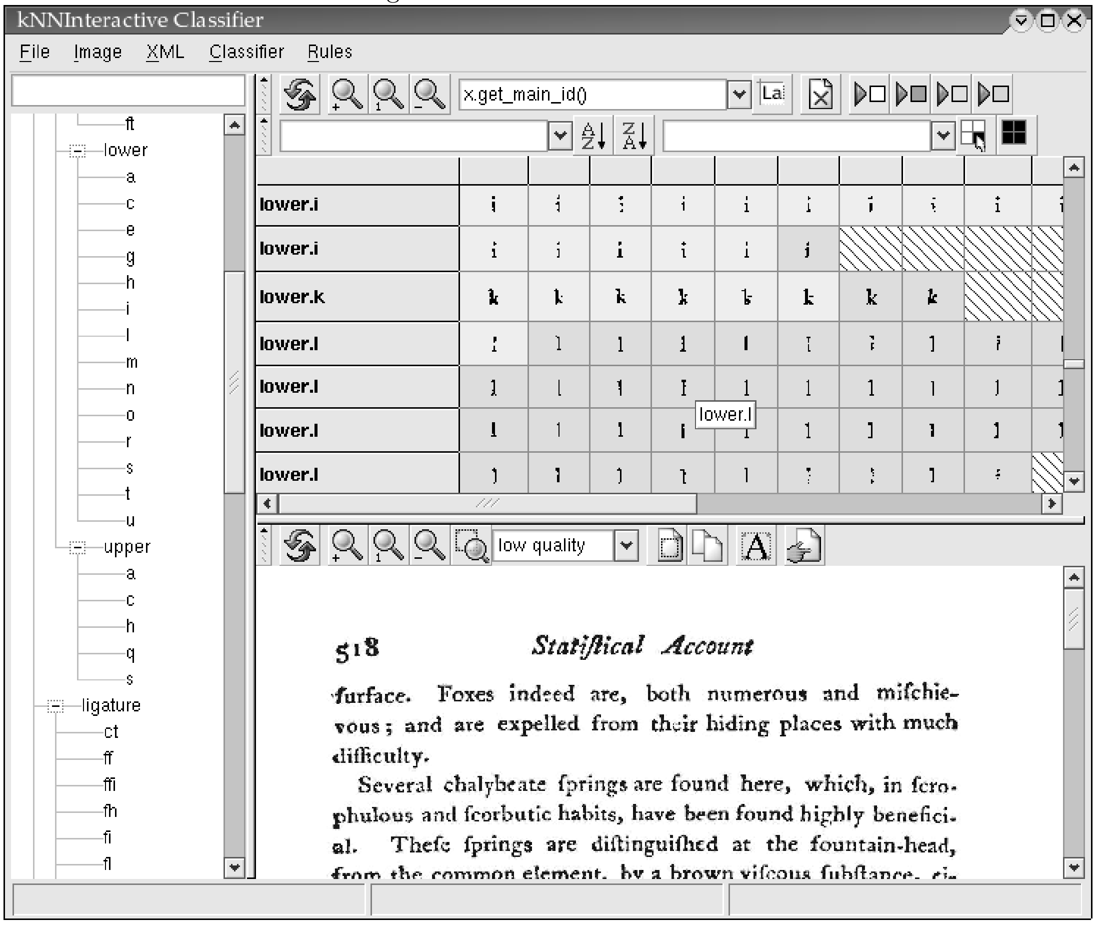
2.3.3 Workflow

Gamera has a workflow infrastructure that allows the end user to easily tie tasks together into a complete recognition system. The workflow engine handles the persistence of data at each step, so that if a particular step fails, it is possible to roll-back and begin in the middle of a process without repeating preceding steps. Again, this encourages the test-and-refine development model. We plan to extend this workflow concept to include a simple visual programming language (signal-flow graph) [11], to make it easier for non-programmers to customize and build their own systems.

2.3.4 Unit-testing

Gamera includes a convenient unit-testing framework for testing plugins or other functions. If no unit tests are written for a given function, stock source images are used with default arguments, and the resulting image is compared to one that has been declared correct. If a step requires more rigorous testing, a custom unit test can be written. In either case, Gamera handles all of the details of saving and

Figure 4: The classifier window.



518 *Statiſtical Account*

ſurface. Foxes indeed are, both numerous and miſchie-
vous; and are expelled from their hiding places with much
difficulty.

Several chalybeate ſprings are found here, which, in ſcro-
phulous and ſcorbutic habits, have been found highly benefici-
al. Theſe ſprings are diſtinguiſhed at the fountain-head,
from the common element, by a brown viſcous ſubſtance, ci-

loading images to disk and verification.

3 Recent developments

That completes the overview of the architecture of Gamera. Some of our recent research includes designed a declarative pattern matching engine, improving the handling of degraded documents, and using clustering techniques to reduce manual training times and provide additional verification. These three subprojects are discussed in detail below.

3.1 Pattern matching engine

A pattern matching engine is included to assist with basic structural analysis problems. Using a logic programming language such as Prolog [12], or perhaps a domain-specific minilanguage, would be suitable for this purpose. However, we felt it would decrease the usability of the system to add yet another programming language to the mix. Therefore, the pattern matching engine is designed so that authors simply write specialized Python functions that add some declarative programming to what is still primarily an imperative approach.

An example rule function for finding periods at the end of sentences is given below.

```
# Find periods at the end of sentences
# (a) function signature
def find_periods(a="dot", b="letter*"):

   # (b) expression
   if (Fudge(a.lr_y) == b.lr_y and
       a.ul_x > b.lr_x and
       a.ul_x - b.lr_x < 20):

      # (c) operation
      period = a.copy()
      period.classify_heuristic(
        'punctuation.period')

      # (d) added, removed
      return [period], []
```

The argument signature (a) defines which symbols will be applied to the function, using regular expressions to match the class name. (Remember, the symbols have already been classified in the previous stage.) The regular expression syntax used is designed to be as convenient and familiar as possible, and is based on what one might find in a Unix shell. In the example, the variable `a` will be unified with glyphs classified as `dot`, and `b` with glyphs of any class beginning with `letter`. The body of the function generally contains an expression (b) and an operation (c). The expression tests for some criteria, such as the relative position of glyphs. The operation will then produce data to make note of the found pattern. Lastly, the function returns two lists of glyphs (d) that should, respectively, be added or removed from the global set of glyphs. If the function causes no direct side-effects (i.e. doesn't directly modify data, but instead returns modified copies), the Gamera GUI is able to provide undo functionality on top of the pattern matching engine, which helps to support the test-and-refine development model.

The time-complexity of the pattern matching engine is improved by storing all the glyphs in a grid index. Only groups of glyphs in the same or adjacent cells are used as candidate groups for arguments to a pattern matching function. The size of the grid cells can be adjusted, to change the amount of adjacency possible within a pattern.

The pattern matching engine has proven to be useful for solving a number of problems, including finding punctuation and grouping parts of cursive words. We plan to improve it to support global optimizations and pattern-ordering facilities. There are also improvements in runtime that could be achieved by indexing the functions by their expressions. Since the approach is so straightforward, however, it is difficult for the pattern matching engine to achieve optimal runtime efficiency. In any case, it remains a very useful rapid prototyping tool, again supporting the test-and-refine model. In addition, the full power of Python can always be used to perform syntactic or structural analysis when the pattern matching tools are too weak or inefficient.

3.2 Character degradation

Most commercial optical character recognition (OCR) systems are designed for well-formed, modern business documents. Recognizing older documents with low-quality or degraded printing is more challenging, due to the high occurrence of broken and touching characters. Gamera includes unique approaches for dealing with both of these problems. These algorithms have been very successful on a set of real-world historical documents.

For the purposes of this paper, a *connected component* (CC) is a set of black pixels that are 8-connected. By definition, characters are *broken* when they are made up of too many CCs, and they are *touching* when they are made up of too few CCs. Broken characters can not be joined simply by the distance of the CCs alone, since two intentionally separate characters can often be closer than the two parts of an accidentally broken character.

3.2.1 Other approaches

Thresholding converts a color or greyscale image to a bi-level image, such that black is used to indicate the presence of ink on the page and white is used to indicate its absence. Improving thresholding by looking for shades of grey in the areas where CCs almost touch, using entropy, can reduce the number of broken characters [13]. However, in many historical documents, the characters are completely broken on the page and intelligent thresholding, since it has no knowledge of the shapes of the target characters, performs poorly.

Active contour models (ACM), or snakes [14], find a vector outline for each symbol using certain constraints on the elasticity of the outline. Unfortunately, ACMs, which were designed for gross shape recognition, perform poorly on the fine details that are required to recognize printed characters.

Post-processing using some kind of language model, including a dictionary or n-grams of a language [15] has also been used to improve the recognition of broken and touching characters. However, such models are less useful for documents containing ancient languages, mixed-languages or a high occurrence of proper nouns.

Therefore, an ideal solution would include knowledge of the individual symbols without requiring a language-specific model.

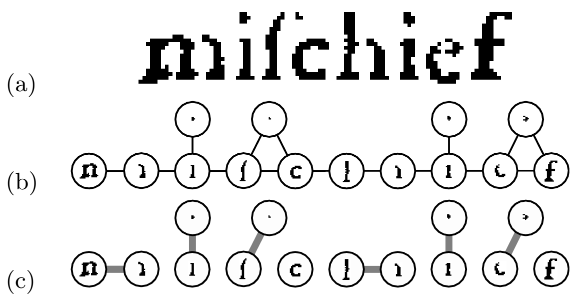
3.2.2 Broken character correction

The goal of the broken character connection (BCC) algorithm is to find an optimal way to join CCs on a given page that maximizes the mean confidence of all characters.

The algorithm begins by building an undirected graph in which each vertex represents a CC in the image. Two vertices are connected by an edge if the border of the bounding boxes are within a certain threshold of distance. Experiments demonstrated that this threshold is best set to $^3/_4$ of the average distance between all bounding boxes. (CCs can also be connected by morphological dilation, though the bounding box method is much faster and produces only slightly less accurate results.) This creates a forest of graphs where each graph is roughly equivalent to a word in the document. Figure 6(b) shows one such graph. A graph representation, rather than a string representation, of connectivity is necessary, since characters can be broken in the x- and/or y-direction and cycles can occur between CCs.

Next, all of the different ways in which the CCs can be joined are evaluated. Every possible connected subgraph is enumerated by performing a depth-first search from each vertex. The subgraphs created from the word in Figure 6(a) are shown in Table 1. To avoid enumerating duplicate possibili-

Figure 6: (a) an original image of a word from the testbed; (b) how the connected components are connected to form a graph; (c) the correct solution.



ties, the vertex v assigned a number N_v, and an edge is traversed from vertex a to b only if $N_b > N_a$. To improve runtimes, the depth of the search is limited to the maximum number of CCs that would typically make up a single broken character. This constant is adjusted automatically based on the amount of degradation in the image and is usually between 3 and 5. The "correctness" of each of the images represented by these subgraphs is evaluated by merging all of its CCs into a single image and sending it to the symbol classifier. The symbol classifier returns a confidence value that indicates how similar the merged image is to known symbols in the database.

Since we are using the k-nearest neighbor (k-NN) classifier [4] for symbol classification, it was most convenient to use a confidence measure based on distance. More elaborate ways of determining confidence, such as analyzing the clustering of symbols within the database, have been suggested, but they do not significantly affect the success of the BCC algorithm.

Once these subgraphs have been evaluated, an optimal combination of them must be found. Each subgraph is represented by a bit-string, in which a 1 indicates the presence of a vertex. The goal is to find combinations of subgraphs such that each vertex is used exactly once. By sorting the list of subgraphs in lexical order by their bitstrings, we can use the following criteria to do this efficiently:

> Given subgraphs P and Q, Q can be added to P if it (a) does not have any vertices in common with P and (b) contains the lowest-numbered vertex that P is missing.

More formally,

> **Criterion A:**
>
> let V be the set of all vertices in the entire graph, and $P \subseteq V,\ Q \subseteq V$;
>
> (a) $\forall v \in Q,\ v \notin P$;
>
> (b) $\forall v \in V,\ N_v < \min(\forall w \in Q,\ N_w) \Rightarrow v \in P$.

The importance of sorting the list of subgraphs is that Q will always be after P in the list, thereby reducing the arity of the search tree. The algorithm starts by setting P to the empty set. All subgraphs Q that meet **Criterion A** are combined with P. The union of P and Q is then used as P recursively, until $P = V$. The mean confidence of all the Q's used in each path of recursion is used to evaluate the combination. The combination with the highest mean confidence is chosen as the correct combination.

The runtime of the combination-finding can be improved by exploiting the fact that each subgraph P has a fixed set of subgraphs Q' that are contiguous in the sorted subgraph list and that meet **Criterion A(b)**. These sets can be found ahead of time in linear times relative to the number of vertices ($O(|V|)$). Once this is done, we've reduced the search space from each P considerably. For example, Table 1 shows the beginning and ending (non-inclusive) indices of Q' for each subgraph in the *from* and *to* columns. In this example, the mean size of Q' is 4.7, which corresponds the mean arity of the search tree. Without this optimization, the mean arity would be $\frac{|V|}{2} = 37$. As an additional optimization, "memoization", borrowed from dynamic programming, is used to store the best result of all subtrees of the search so that identical subtrees do not need to be traversed multiple times.

While a full runtime complexity analysis of the entire BCC algorithm is beyond the scope of this paper, the asymptotic upper bound is $O(n \ln n)$, where n is the number of vertices in the graph. However, when there are no cycles, the runtime is reduced to roughly $O(kn)$, where k is the maximum size of the subgraphs (usually $3 \leq k \leq 5$).

3.2.3 Touching character correction

To deal with touching characters, we have an entirely different approach that was first used for optical music recognition [16]. The symbol classifier is trained with examples of touching characters, which are given a class name beginning with the special token `_split`. When the classifier later matches an unknown to an example starting with `_split`, it will perform a splitting operation on the connected component, and then recursively classify the results. The splitting operation can be any function that examines a connected component and splits it into multiple connected components through some kind of heuristic process.

For example, we have implemented primitive `splitx` function that splits a connected component by finding a minimum projection on the x-axis near the center of the connected component. If we let p_x be the size of the projection (number of black pixels)

Table 1: The bitfields and jumps created from the graph in Figure 6(b). The bits have been reversed to match the way the characters are read (left-to-right).

#	n	ı	,	ı	ſ	.	c	l	ı	t	,	c	,	f		
N_v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	from	to
0	■														7	18
1	■	■													18	28
2	■	■	■												28	34
3	■	■	■	■											34	41
4	■	■	■		■										28	34
5	■	■		■											18	28
6	■	■		■	■										18	28
7		■													18	28
8		■	■												28	34
9		■	■	■											34	41
10		■	■	■	■										41	45
11		■	■		■										28	34
12		■	■		■	■									28	34
13		■	■		■		■								28	34
14		■		■											18	28
15		■		■	■										18	28
16		■		■	■	■									18	28
17		■		■	■		■								18	28
18			■												28	34
19			■	■											34	41
20			■	■	■										41	45
21			■	■	■	■									45	49
22			■	■	■		■								41	45
23			■		■										28	34
24			■		■	■									28	34
25			■		■	■	■								28	34
26			■		■		■								28	34
27			■		■		■	■							28	34
28				■											34	41
29				■	■										41	45
30				■	■	■									45	49
31				■	■	■	■								49	54
32				■	■		■								41	45
33				■	■		■	■							41	45
34					■										41	45
35					■	■									45	49
36					■	■	■								49	54
37					■	■	■	■							54	60
38					■		■								41	45
39					■		■	■							41	45
40					■		■	■	■						41	45
41						■									45	49
42						■	■								49	54
43						■	■	■							54	60
44						■	■	■	■						60	66
45							■								49	54
46							■	■							54	60
47							■	■	■						60	66
48							■	■	■	■					66	67
49								■							54	60
50								■	■						60	66
51								■	■	■					66	67
52								■	■	■	■				67	71
53								■	■	■		■			66	67
54									■						60	66
55									■	■					66	67
56									■	■	■				67	71
57									■	■		■			66	67
58									■	■		■	■		66	67
59									■	■		■		■	66	67
60										■					66	67
61										■	■				67	71
62										■		■			66	67
63										■		■	■		66	67
64										■		■	■	■	66	67
65										■		■		■	66	67
66											■				67	71
67												■			71	73
68												■	■		73	74
69												■	■	■	74	74
70												■		■	71	73
71													■		73	74
72													■	■	74	74
73														■	74	74

Figure 7: Example of touching characters being split using a projections-based split algorithm.

in column x, and d_x be the distance from the center along the x-axis ($|\frac{w}{2} - x|$), the score for each column is determined with the Euclidean distance equation:

$$s = \sqrt{{p_x}^2 + {d_x}^2} \qquad (1)$$

The column with the minimum score is selected as the split point. Figure 7 shows an example of splitting a connected component, classified as `_split.splitx`, using this method. Of course, more sophisticated ways of splitting are possible. For example, we have also implemented an algorithm that removes vertical lines from compound musical structures in order to separate their components.

The runtime complexity of this approach is dependent on the underlying symbol classifier and the splitting operation. The splitting operation defined above runs in linear time.

3.2.4 Results

To evaluate the BCC algorithm, we used the Statistical Accounts of Scotland [17]. This collection of census-like data was printed in 1799, with reused metal type on wooden blocks. The age of the paper, combined with the low-quality type and press-work, presents challenges for OCR. The typeface is characterized from modern business documents by the use of the "long s" ($\int$) and a high occurence of ligatures **ff**, **ffi**, **ffl**, etc.) Table 2 shows the distribution of the types of characters in the collection. The manually-generated groundtruth data also makes this collection valuable for research.

Table 2: Distribution of character types in the sample data. Note that failure to deal with broken and touching characters gives a maximum possible accuracy of 81.3%.

complete characters	81.3%
broken characters	10.7%
legit. broken characters (**i**, **j**, **;**, **:** etc.)	6.2%
touching characters	1.8%

The results below were obtained by training the classifier using five pages, and then testing the algorithm on five additional unseen pages with similar typeface.

If the symbol classifier only has knowledge of the complete characters in the image, BCC correctly finds 71% of the broken characters in our test data. By training the symbol classifier with examples of broken characters that were manually identified, BCC correctly finds 91% of the broken characters.

BCC also performs well with legitimately broken characters, such as **i**, **j**, **;** and **:**. By training the symbol classifier with examples of each of these characters, BCC was able to find and join 93% of the legitimately broken characters. This renders further procedural programming of heuristic rules (such as to attach **ı**'s to dots) unnecessary. Therefore, it easy to support new character sets that have other legitimately broken characters, such as the Greek majuscule xi (**Ξ**).

As for touching characters, our approach catches and correctly splits 93% of the touching characters in the dataset. Note, however, that touching characters had a much lower occurrence than broken characters.

3.3 Improving learning classifiers with clustering

Learning classifiers provide a flexible basis for the creation of document recognition tools that adapt to the specific set of characters of symbols present in a document [18]. Unfortunately, the manual labeling of thousands of examples that is required to train a learning classifier is time-consuming and error-prone. In the case of cultural heritage materials, which may be degraded or in obscure languages, training may require specialized knowledge, making it more difficult to find users capable of training a system [1]. Additionally, verifying the training data after labeling is important to ensure the accurate performance of the classifier. Finally, the automatic evaluation of the performance of a learning classifier is required for large digitization projects. This section presents the use of clustering for the prelabeling of symbols to shorten training time and for the verification of the data after training.

3.3.1 Clustering Overview

Clustering is the organization of a collection of input patterns into clusters based on similarity. In the context of document recognition, a clustering algorithm would be presented with the symbols from a document and would, ideally, group all of the same symbols together. Clustering is a large field with many different goals and techniques (see [19] for a review of clustering techniques and applications). In this application of clustering to multi-lingual document recognition, graph-theoretic clustering was chosen because it does not require the target number of clusters to be known, is deterministic, and has reasonable algorithmic complexity for the size data considered [20, 19].

3.3.2 Graph-Theoretic Clustering

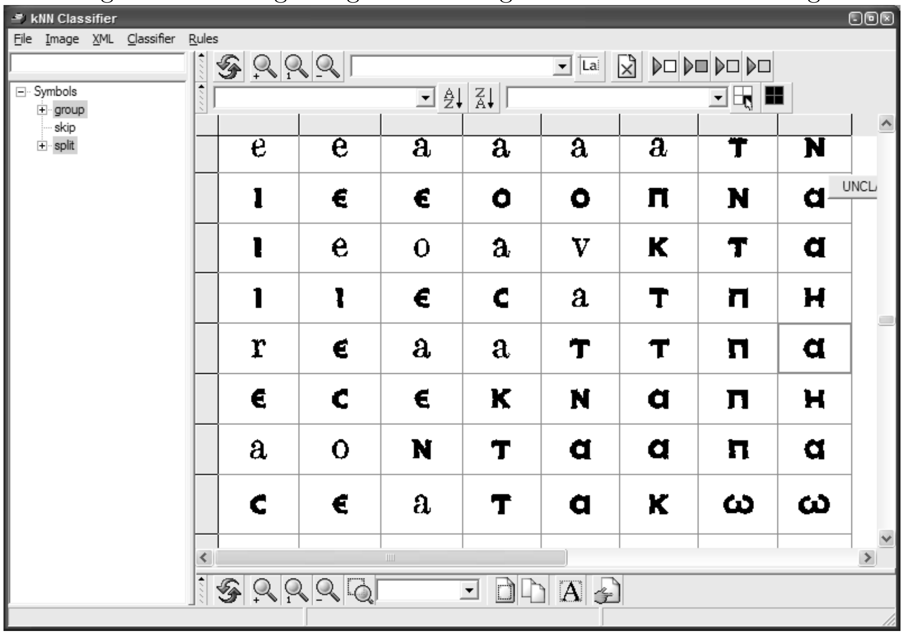
Graph-theoretic clustering operates by creating a minimum spanning tree (MST) of a set of points and deleting edges in the graph based on some criterion [20]. The sub-graphs created by the deletion of the edges represent the clusters. In this application, the point set is created by segmenting an image into symbols and measuring the distance between all of the unique pairs of symbols. The distance measurement, which operates on a set of features representing the symbol, used in these experiments is either simple Euclidean or Manhattan. The criterion used for the deletion of edges is the standard deviation of the current edge weight to the mean of the edge weights of the edges within a local window. Experimentation reveals that using a standard deviation of above 1.5 and window size of 2 to 4 nodes from the current edge yields good results on a variety of documents.

3.3.3 Pre-classification clustering

Instance-based learning classifiers, like the nearest-neighbor classifier, leverage the knowledge of domain-experts by allowing them to train the classifier by identifying (labeling), a set of symbols that the classifier uses to identify subsequent symbols. This allows the classifier to be used to identify almost any symbols without additional software development. Unfortunately, the time required to train learning classifiers can be a major drawback.

At the beginning of the training process the system, by definition, has no knowledge of the symbols that are about to be labeled. It is not possible, therefore, to present the symbols in a way that makes their relationships apparent to the user, making the task of training take significantly longer than if the symbols were already grouped. Figure 8 shows the initial state of the Gamera training tool.

Figure 8: The beginning of a training session without clustering.



Using clustering to group the symbols together before presenting them to the user can reduce the training time, however. This reduces the problem of training to labeling groups of symbols and has the potential to reducing training mistakes. Figure 9 shows the initial state of the Gamera training tool when pre-labeling clustering is used.

3.3.4 Post-Training Verification

The effectiveness of a learning classifier is dependent on the quality of the training set. Clustering can be used to help a user verify the manual labeling of symbols. Each group of manually trained symbols can be clustered and the results presented to the user. Because any incorrectly labeled symbols will appear as a separate cluster in the results it is much easier for the user to identify mistakes than if presented with all of the symbols from a group without clustering. It is important to note that this verification process cannot be fully automatic, even if the clustering algorithm performed without errors, because a user may choose to label visually dissimilar glyphs as the same symbol. For example, the same letter in multiple typefaces might be labeled as one symbol. This flexibility in the training is an important aspect of many learning algorithms and should be preserved.

4 Conclusion

The Gamera system is well on its way to being a complete framework for the development of recognizers for a broad range of historical documents. Our own use of the system has helped us find weaknesses in the framework that we've subsequently improved. We are also pleased by how much more efficient and less frustrating programming in the Gamera framework is relative to building standalone applications from scratch. Our success in recent applications with real-world data is a testament to the utility of our approach.

Acknowledgments

The second phase of the Levy Project is funded through the National Science Foundation's Digital Library Initiative 2 (Award #9817430), an Institute of Museum and Library Services National Leadership Grant, and support from the Levy family.

We also wish to thank the rest of the Digital Knowledge Center staff, (Teal Anderson, G. Sayeed Choudhury, Tim DiLauro, Mark Patton and Cynthia York), for their assistance with this paper.

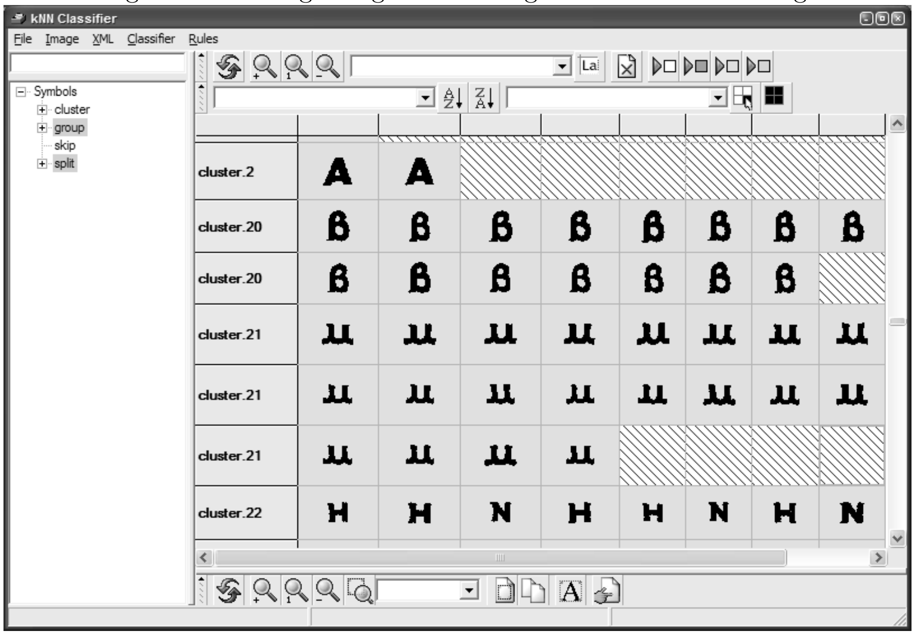
References

[1] Droettboom, M., K. MacMillan, I. Fujinaga, G. S. Choudhury, T. DiLauro, M. Patton, and T. Anderson. 2002. Using the Gamera framework for the recognition of cultural heritage materials. *Joint Conference on Digital Libraries.* 11–7.

[2] Rossum, G. 2002. Python language reference. F. L. Drake, Jr., ed. `http://www.python.org`

[3] Berehzny, L., J. Elkner, and J. Straw. 2001. Using Python in a high school computer science

Figure 9: The beginning of a training session with clustering.



program: Year 2. *International Python Conference*. 217–23.

[4] Cover, T. and P. Hart. 1967. Nearest neighbour pattern classification. *IEEE Transactions on Information Theory*. 13(1): 21–7.

[5] Holland, J. H. 1975. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor.

[6] Schemenauer, N. 2002. *A simple neural network library* (`bpnn.py`). Computer software. `http://arctrix.com/nas/`

[7] Hoos, H. H. and K. Hamel. 1997. GUIDO music notation version 1.0: Specification part I, Basic GUIDO. Technical Report 20, Technische Universität Darmstadt.

[8] Dunn., R. *wxPython toolkit*. Computer software. `http://www.wxpython.org`

[9] Jähne, B., H. Haußecker, and P. Geißler. 1999. Reusable software in Computer Vision. *Handbook on Computer Vision and Applications*. New York: Academic Press.

[10] Bøe, S., T. Lønnestad, and O. Milvang. 1998. XITE: X-based image processing tools and environment: User's manual, version 3.4. Technical Report 56, Image Processing Laboratory, Department of Informatics, University of Oslo.

[11] Puckette, M. 1988. The Patcher. *International Computer Music Conference*. 420–9.

[12] Clocksin, W. F. and C. S. Mellish. 1981. Programming in PROLOG. Berlin: Springer.

[13] Trier, Ø. D. and A. K. Jain. 1995. Goal-directed evaluation of binarization methods. *IEEE Transactions on Pattern Analysis & Machine Intelligence*. 17(12): 1191–201.

[14] Kass, M., A. Witkin, and D. Terzopolous. 1987. Snakes: Active contour models. *International Conference on Computer Vision*. 259–68.

[15] Harding, S. M., W. B. Croft, and C. Weir. 1997. Probabilistic retrieval of OCR degraded text using N-grams. *European Conference on Digital Libraries*. 345–59.

[16] Fujinaga, I., B. Alphonce, B. Pennycook, and K. Hogan. 1991. Optical music recognition: Progress report. *International Computer Music Conference*. 66–73.

[17] *Statistical Accounts of Scotland*. 1799. `http://edina.ac.uk/statacc/`

[18] MacMillan, K., M. Droettboom, and I. Fujinaga. 2001. Gamera: A Python-based toolkit for structured document recognition. *Tenth International Python Conference*.

[19] Jain, A. K., M. N. Murty, and P. J. Flynn. 1999. Data Clustering: A Review. *ACM Computing Surveys*. 31(3): 264–323.

[20] Zahn, C. T. 1971. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*. 20: 68–86.