Kanji Character Recognition Techniques

Etai Klein

June 8, 2014

Abstract

In this project, I attempt to create an accurate stroke-order-agnostic classification system for Japanese and Chinese characters (Kanji). I compare accuracy across 3 different classification methods across 2 data sets and 2 levels of Japanese Kanji features. This survey is the first half of a project which will result in the creation of an app to help students of Japanese practice their handwriting. I coded the project in Java using a modular design.

1 Introduction

Current Kanji classification methods inaccurately classify strokes that are out of order or combined. This causes problems for both new students and native speakers. In my project I address this by surveying classification techniques to find the most accurate stroke-order agnostic classification system.

2 Materials and Methods

In this section, I discuss the datasets, feature levels, and classification methods used in this survey.

2.1 datasets

I used two data sets for this project:

2.1.1 CASIA Online and Offline Chinese Handwriting Databases

This is a Chinese handwriting database which can be found at `<http://www.nlpr.ia.ac.cn/databases/handwriting/Home.html>`. I used their Chinese Handwriting Recognition competitions dataset which contains 60 tokens of almost 4000 types of Kanji.

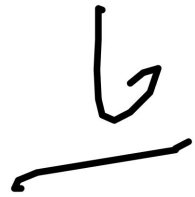

Figure 1: An example of the CASIA database of the Kanji (ue)

The data was an encoded series of bytes and had to be decoded. Each file contained almost 4000 Kanji with the values: size (bytes), GBK tagcode, number of strokes, concatenated xycoordinates, and an end marker. The decoding script can be found in `ChineseOnlineHandwritingToJPG.java`.

The Kanji's name was tagged using GBK, an encoding standard like unicode which is used in China, and required translation. The translation script can be found in `ChineseCharacterLookup.java`wherein I parse the html from this online table `<http://www.cs.nyu.edu/~yusuke/tools/unicode_to_gb2312_or_gbk_table.html>`

2.1.2 Collected data

I created an app to collect data from students learning Japanese (as per the app's target audience). The app can be found in the project `datacollection`. The basic idea was that I prompted users to write a character displayed at the top of the screen. When users hit the next button, their data was saved to dictionaries for compressed bitmaps and strokes. Then, when the user hit the save button, the dictionaries were saved to the phone to be collected later.

I pre-processed the Collected data by scaling each image to the same size.

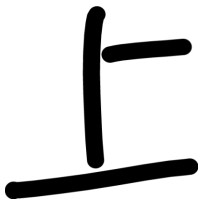

Figure 2: An example of the data I collected of the Kanji

2.2 Feature levels

classification essentially is about finding the difference between two or more items (like Kanji). One way to computationally classify an unknown Kanji is to quantify some features about the Kanji and compare it to other Kanji to find the most similar one. Humans are much worse at choosing features than machines.

Below, I outline the features I chose to test and how I used a program to make better feature choices.

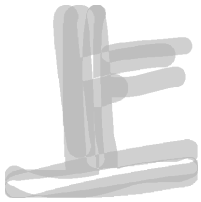

Figure 3: An example of the averaged Collected data for the Kanji

2.2.1 Pixel Level

The simplest way to do any Optical Character Recognition is to use pixels as a baseline. I used the lightness of each pixel in my images as my features for classification.

Below I've included my distance method for pixel level analysis:

```
/**
 * distance
 *
 * @summary Computes a score based on how different one kanji is from this kanji
 *
 * @param otherKanjiPixels - The pixels of the Kanji to Compare to
 * @return sum - the difference score
 *
 * @pseudocode
 * 1. loop through each kanji's pixel array
 * 2. sum the difference between pixels
 *
 */
public int distance(int[][] otherKanjiPixels){

  double sum = 0;

  for (int row = 0; row < averagePixels.length; row++) {
    for (int col = 0; col < averagePixels[0].length; col++) {
      //sum the difference between each pixel
      sum += Math.abs((double) averagePixels[row][col] - (double)otherKanjiPixels[row][col
          ]);
    }
  }

  return (int)sum;
}
```

2.2.2 Stroke Level

One way to classify Kanji is to look at the sume of their strokes, the building blocks of Kanji. There are 2,136 Joyo (daily use) Kanji as defined by the Japanese Ministry of Education in 2010. Kanji are made from

groups of Bushu (Radicals), which are in turn made from ordered groups of strokes. There are 214 total Bushu. The daily use Kanji all have fewer than 30 strokes.

strokes, unlike pixels, don't have obvious features. I used the following features:

- stroke length (start point to end point)
- stroke angle (start point to end point)
- direction between strokes (end point to start point)

Below I've included my distance method for stroke level analysis:

```
/**
   * distance
   *
   * @summary Computes a score based on how different one kanji is from this kanji
   *
   * @param kanji - The pixels of the Kanji to Compare to
   * @return sum - the difference score
   *
   * @pseudocode
   * 1. loop through each kanji's strokes
   * 2. Sum the difference between lengths, angles, and moves
   *
   * @note: moves is weighted by 100 since it is a more useful feature
   *
   */
  public double distance(StrokeKanji kanji) {

    int maxStrokes = 30;
    int weight = 100;

    int sum = 0;

    int i = 0;
    while (i < maxStrokes){

      sum+= Math.abs(this.avelengths[i] - kanji.lengths[i]);
      sum+= Math.abs(this.aveangles[i] - kanji.aveangles[i]);

      //moves are the difference between strokes, so there will be one less maximum move
          than maximum stroke
      if (i < maxStrokes-1){
        sum+= weight * Math.abs(this.avemoves[i] - kanji.moves[i]); //weighted
      }

      i++;
    }

    return sum;
  }
```

2.3 classification Methods

The basic idea for any classification technique is to score all possible classifications based on the test kanji's features and pick the best score. In order to have something to score the kanji against, I split my data sets into training and testing sets.

2.3.1 K-Nearest-Neighbors

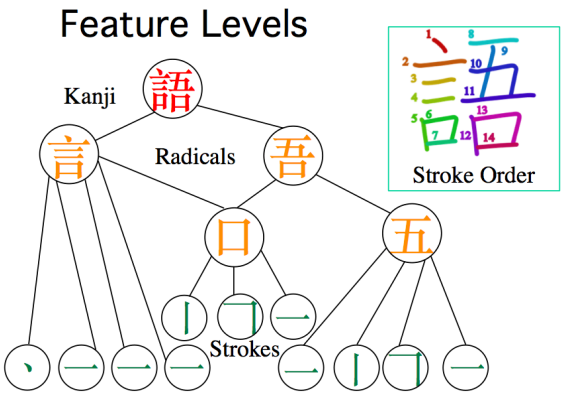

Figure 4: The Feature levels of a Kanji beyond the pixels level.

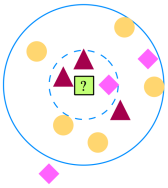

Figure 5: If K is 3 (the smaller circle) then we would classify the unknown shape as a red triangle. If K is 10 (the larger circle) then we would classify the unknown shape as a yellow circle

K-Nearest-Neighbors (KNN) works by assigning scores to all Kanjis in the training datasets. Then I score the training dataset and classify it as the mode of the K best scores.

2.3.2 Gaussian Distribution

The Guassian distribution works by computing the average and the standard deviation of every feature for every Kanji type. Then I classify an unknown Kanji as the Kanji with the lowest sum z-score.

2.3.3 Univariate Distribution

The Guassian distribution works by computing the average of every feature for every Kanji type to compute a single average Kanji for each type. Then I score each average Kanji and label the unknown Kanji as the average Kanji with the best score.

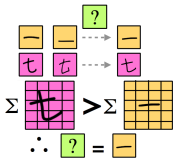

Figure 6: The sum of the pixels z-scores for the unknown shape lower for the yellow box.

2.3.4 WEKA Multivariates

WEKA (Waikato Environment for Knowledge Analysis) is a data analysis and machine learning tool. I used WEKA's java plugin to calculate multivariate data. A multivariate classifier uses complex algorithms involving the relationship between features for classification. The two WEKA tests I was interested in were IBK, a K-Nearest-Neighbors classifier and SMO, a Sequential Minimal Optimization algorithm for training a support vector classifier (which I am choosing to think of as a multivariate distribution classifier).

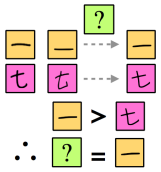

Figure 7: The unknown shape is closer in similarity to the average yellow box on a pixel by pixel level.

3 Code Structure

I used a modular design for this project such that I could easily swap out any dataset, feature level, classification method for another. I organized my classes into the following packages:

- `classifierInterfaces`- Interfaces for the `PixellevelClassifier`and `strokelevelClassifiers`.
- controller - The main controller from where you can run all tests

- `KanjiClasses`- Defines a `Kanji`class and its specialized subtypes, `PixelKanji`and `strokeKanji`.
- `pixellevelClassifiers`- defines the `KNN`,`Gaussian`and `WEKA`classifiers.
- `strokelevelClassifiers`- defines the `KNN`,`Gaussian`and `WEKA`classifiers.
- `utils`- Contains classes related to image and file processing.

4 Results

The data that I collected outperformed the CASIA database data on every test. On average, stroke-level classification outperformed Pixel-level classification.

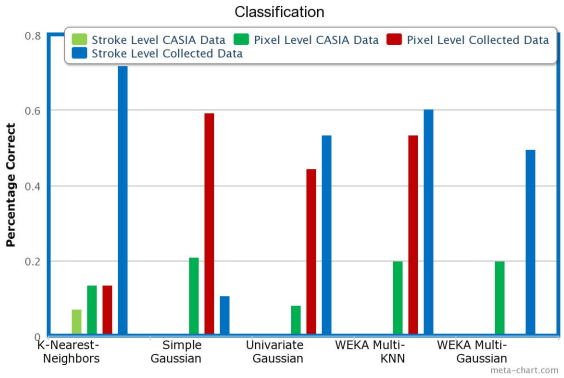

Figure 8: Classification results across feature levels and data sets.

5 Discussion

Native speakers and students have different difficulties in learning Japanese. Native speakers aren't careful about seperating their strokes. Students are write rigidly and frequently get the stroke order wrong. The difference can be seen in `Figure1` and `Figure2`. It is unsurprising that neat handwriting makes classification easier.

Pixel-level data was difficult to use since there were 400*400 features for each image. If I used the whole CASIA dataset, I would have to process over 35 billion features. For this reason I used a reduced dataset for the CASIA data.

To get stroke-level data I had to re-process the original files for both datasets. The Collected data contains 103 datapoints per file, while the Collected data contains almost 4000 datapoints. These will be added to the Results file later when they finish processing.

Results show promise for stroke-level data collection. However, the stroke-level data is not order-agnostic. In the `FutureWork`section I talk about my plans for future order-agnostic stroke-level algorithms.

5.1 Future Work

The next algorithm I want to implement is a bootstrapping algorithm that classifies strokes into radicals and radicals into Kanji. This skates is stroke-order-agnostic since it has a good probability of classifying the correct radical even if the stroke order is wrong. Likewise, it has a good probability of classifying the correct Kanji even if the radical order is wrong!

There are prior probabilities regarding the spatial ordering of the radicals I can use to inform my probabilties i.e. east, west or enclosure.

Another idea is to force stroke-order-agnostic classification is to do stroke-level preprocessing. For example, knowing how many strokes should be in a Kanji I can classify all points the user's pen passes through and recreate the strokes in post-processing. This will be informed by the rules around drawing strokes in Japanese.