Kanji Character Recognition Techniques

Etai Klein
Computational Neuroscience 14F: Richard Granger

November 21, 2014

Abstract

Current recognition algorithms are much worse than human brains at recognizing handwriting. Some advantages the brain has over computers are feature analysis, pattern recognition, parallel computation and contextual analysis. Neuroscientists have modeled brain functions and contributed many useful algorithms to the field of computer science. This paper seeks to compare Neural Networks and Computer Engineered methods of classification and suggest improvements to each method.

1 Introduction

Computational classification algorithms and Neural Networks originate in computer science and neuroscience respectively. In recent history the two fields have taken more influence from each other for a variety of applications. Neural Networks can be very powerful and new ones, like the Cortico-Striatal Loop (CSL), are still being discovered and derived into algorithms (Chandrashekar, Ashok, and Richard Granger). How do current classification algorithms match up against current neural networks? How can either be improved?

In this study, I evaluate the effectiveness of two methods of recognizing Japanese Characters (*Kanji*) and I analyze the effects of modifying both algorithms. The first method is a classification algorithm, K-Nearest Neighbors (KNN), and the second is a neural network, Cortico-Striatal Loops. I hypothesize that both algorithms would perform equally well on the data, as they did in the seminal paper on CSL, and that adding a hidden layer of feature weights will improve the accuracy of both methods (Chandrashekar, Ashok, and Richard Granger).

2 Data

2.1 Kanji

Kanji are Japanese adopted Chinese logographic characters. Kanji are a challenge for classification since they are numerous and share subcharacters. There were 2,136 *joyo* (daily use) *kanji* 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*.

I gathered a total dataset of 4 tokens of 103 types of *Kanji* for this project.

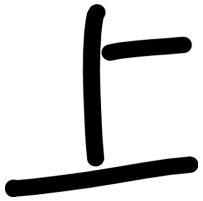

Figure 1: A token of the *kanji*

2.2 Feature Analysis

Kanji Classification algorithms use features of the data and patterns of those features to classify tokens. Human brains probably use a mix of visual features, patterns and contextual features to classify tokens.

Online classification allows information about the kanji down to the stroke level to become accessible (fig. 2). This basic structural information informs the

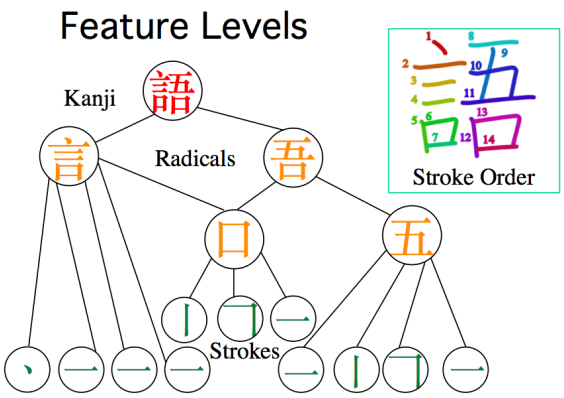

Figure 2: The Feature levels of a *kanji*

features for classification and is key to the success of a character recognition algorithm. Below I've included my distance method for stroke level analysis.

```
/**
   * distance | weighted
   *
   * @summary Computes a score based on how different one kanji is from this kanji using
       weight inputs
   *
   * @param kanji - The Kanji to compare to
   * @return sum - the difference score
   *
   * features:
   * stroke length (start point to end point)
   * stroke angle (start point to end point)
   * angle between strokes (end point to start point)
   * distance of stroke midpoint from kanji center (midpoint to center)
   * number of strokes
   *
   * @pseudocode
   * 1. loop through each kanji's strokes
   * 2. Sum the difference between lengths, angles distances and number of strokes
   *
   */
  public double distance(StrokeKanji kanji, double weightLengths, double weightAngles,
      double weightDistances, double weightMoves, double weightStrokes){

    int sum = 0;
    int i = 0;

    while (i < numstrokes ){

      //all weights default to 1
      sum+= weightLengths * Math.abs(this.avelengths[i] - kanji.lengths[i]);
      sum+= weightAngles * Math.abs(this.aveangles[i] - kanji.aveangles[i]);
      sum+= weightDistances * Math.abs(this.distances[i] - kanji.distances[i]);
      sum+= weightMoves * Math.abs(this.avemoves[i] - kanji.moves[i]);

      i++;
    }

    sum+= weightStrokes * (numstrokes - kanji.numstrokes);

    return sum;
  }
```

3 Classification Algorithms

3.1 K-Nearest Neighbors

K-Nearest-Neighbors (KNN) is widely used for classification and pattern recognition. I chose this algorithm because it is reliable even for a dataset with many types and few tokens.

KNN classifies a test kanji by assigning scores to all *kanji*s in the training datasets (based on the distance algorithm above). The test kanji is classified as mode of the K best scores. I've included my KNN class below:

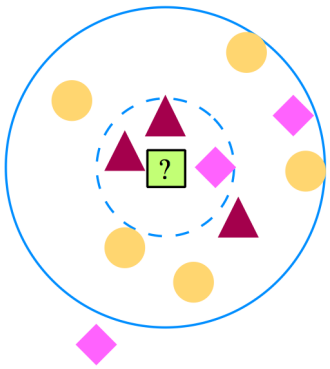

Figure 3: If K is 3 (the smaller circle) then we would classify the unknown shape as a red triangle. If K is 10 (the larger circle) then we would classify the unknown shape as a yellow circle

```
public class StrokeKnn implements KNNInterface{


  ArrayList<StrokeKanji> vectorSpace;
  int kNearest;

  public HashMap<Character, Integer[]> counter = new HashMap<Character, Integer[]>();

  //constructor initializes new vector space and k-nearest neighbors k value
  public StrokeKnn(int k){
    //a list of data points
    vectorSpace = new ArrayList<StrokeKanji>();
    //how many data points do we look at to determine a new point's classification?
    kNearest = k;
  }


  /************************** METHODS **************************/



  /**
   * train -
   * adds a Kanji to the list of kanjis
   * @param Kanji k - the kanji to train
   *
   */

  public void train(Kanji k){
    vectorSpace.add((StrokeKanji)k);
  }

  /**
```

```
 * classify -
 * labels a DataPoint based on the labels of the k-nearest neighbors of a vector
 *
 * @param ArrayList<StrokeKanji> vectorspace (model)
 * @param int (nearest neighbors)
 * @param StrokeKanji (Datapoint to be classified)
 * @return Strokekanji (Datapoint labeled)
 *
 * @pseudocode:
 * 1. Compute the vector from the datapoint
 * 2. List the k vectors with the smallest distance to the datapoint
 * 3. Label the datapoint the modal label of the list
 *
 */

public StrokeKanji classify(ArrayList<StrokeKanji> vectors, int knear, StrokeKanji kanji){

  //1. count distances

  //for each new vector
  for (StrokeKanji k : vectors){
    //get the new vector's distance value
    k.distance = (int) k.distance(kanji);
  }


  //2. sort by distance

  //sort the vectors by distance
  Collections.sort(vectors, new Comparator<StrokeKanji>(){

    public int compare(StrokeKanji v1, StrokeKanji v2) {
      if (v1.distance > v2.distance){
        return 1;
      }
      else if (v1.distance < v2.distance){
        return -1;
      }
      else {return 0;}
    }

  });


  //3. get the k closest vectors

  //intialize a map to hold the number of times each label appears in the knearest
  HashMap<Character, Integer> labelCount = new HashMap<Character, Integer>();

  // run through the nearest vectors
  for (int i = 0; i< knear; i++){

    //get the current label
    char currentLabel = vectors.get(i).label;

    // if we've seen the label before, increase the value by 1
    if (labelCount.containsKey(currentLabel)){
      labelCount.put(currentLabel, labelCount.get(currentLabel) + 1);
    }
    // if we haven't seen the label before, set the value to 1
    else{labelCount.put(currentLabel, 1);}
  }


  //4. find the most popular label
```

```
    //get all the labels in the set
    Set<Character> keys = labelCount.keySet();
    //initialize our max value with the first one
    Character max = keys.iterator().next();

    for (Character key : keys ){
      if (labelCount.get(key) > labelCount.get(max)){
        max = key;
      }
    }


    //5. set the label

    Character original = kanji.label;
    kanji.label = max;

    return kanji;
  }
```

4 Neural Networks

4.1 Cortico-Striatal Loop

Cortico-Striatal Loop (CSL) refers to two telencephalic structures, the cortex and striatum, which are thought to perform unsupervised hierarchical categorization of both static and changing signals. The theory goes that there are two subcircuits: the core, a highly topographic, successive, hierarchical clustering mechanism, and the matrix, a highly non-topographic differentiation mechanism ((Chandrashekar, Ashok, and Richard Granger). An algorithm derived from this mechanism results in successive subclustering that can quickly and accurately classify either static or changing signals.

I choose this particular algorithm because it classifies data as well as KNN, but with lower time and space complexities. This is particularly impressive since its an unsupervised network, while KNN is supervised.

CSL can classify a test kanji by building a tree from a training dataset through successive unsupervised subcategorization and then navigating it to the leaf closest to the test kanji (see the distance algorithm above).

Implementing CSL involves two basic design choices: clustering and classifying. I decided to use K-means for clustering and tree descent for classifying since they are simple to implement.

I've included my CSL class below:

```
  /**
   * runCSL
   * builds a tree from all training kanji and classifies all test kanji
   *
   * @param test - the kanji to classify
   * @param train - the kanji to use in the distribution
   * @param fileType
   * @throws Exception
   *
   * @pseudocode:
   * 1.add each kanji to the root
   * 2.cluster in numClusters new nodes
   * 3.if any nodes are not uniformly labeled, repeat 2-3
   * 4.traverse the tree to classify
   *
   */
  public void runCSL(ArrayList<StrokeKanji> trainKanjis, ArrayList<StrokeKanji> testKanjis,
      numClusters){
```

```
    //1. add each kanji to the data for the root of the tree
    for (StrokeKanji kanji : trainKanjis){
      T.getRoot().getData().add(kanji);
    }

    //add the root node to the queue
    Q.add(T.getRoot());

    //iterate until the Q is empty (all leaf nodes have uniform labels)
    while (!Q.isEmpty()) {
      //pop the top node
      CSLNode currentNode = Q.poll();
      //do not cluster if the node is uniform
      if (isUniform(currentNode)) {continue;}
      //run kmeans for clustering
      HashMap<Instance, ArrayList<StrokeKanji>> clusters = kMeans(currentNode, numClusters
          );

      for (Instance centroid : clusters.keySet()){
        //create a new child node and explicitly set data
        CSLNode childNode = T.new CSLNode(T.depth);
        childNode.setCentroid(centroid);
        childNode.setData(clusters.get(centroid));
        childNode.label = clusters.get(centroid).get(0);
        //set parent/child relationship for traversal
        childNode.parent = currentNode;
        currentNode.children.add(childNode);
        //add the child node to the queue
        Q.add(childNode);
      }
      //mark the tree's depth
      T.depth++;
    }

    //testing
    classify(testKanjis);
}
```

5 Neural Network Inspired Improvements

A common factor linking the more successful Neural Networks is the ability to scale or reduce the weights applied to features. This is often implemented as a hidden layer of nodes that can adjust feature weights.

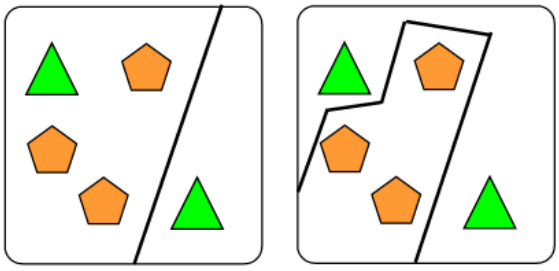

Figure 4: Perceptron [- hidden nodes] vs. Backpropogation [+ hidden nodes]. The Backpropogation algorithm is able to cleanly separate the two types of inputs while the Perceptron is not.

Adding a layer of hidden nodes can be extremely powerful. For example it is the main difference between the Perceptron Neural Network which can only classify linearly seperable data and the BackPropogation

Neural Network which can classify more robustly.

I added a very simple hidden layer multiplier to my features for KNN and CSL. I predict this will improve the accuracy rate of both algorithms above. The basic idea is that I include weights in the distance algorithm and rerun the classification with increasing weights until I find a local maximum of correct identifications. The distance algorithm above is already parameterized to take weights. The algorithm for updating the weights can be seen below:

```
/** RecursivelyClassify
 *
 * Reruns the classify algorithm to find the most correct local maximum of weights
 *
 * @param CSLTree T - the Tree containing all the training data
 * @param Callable<Integer> classify - the classification method
 * @param ArrayList<StrokeKanji> kanjis - the test data
 * @param int[] weights - the weights for this set of classifications (init 0)
 * @param int bestcorrect - the most correct classifications for any run
 * @param int[] bestweights - the weights resulting in the best classification
 * @param double weightValue - the weight multiplier
 * @param double weightDiminisher - reduces the weight multiplier
 *
 * @pseudocode:
 * 1. run the classification algorithm
 * 2. return if you didn't score better
 * 3. run again increasing each weight by a diminishing factor
 *
 * @return
 */

public void recursivelyClassify(CSLTreeWeka T, Callable<Integer> classify, ArrayList<
    StrokeKanji> kanjis, int[] weights, int bestcorrect, int[] bestweights, double
    weightValue, double weightDiminisher){

  //1. classify
  int correct = classify(T, kanjis);
  //2. return if you didn't score better
  //can switch to >= to increase the chance of finding another local maximum
  if (correct > bestcorrect){
    bestcorrect = correct;
    bestweights = weights;
    //3a. run again increasing each weight
    for (int i = 0; i < weights.length; i++){
      //3b. by a diminishing factor
      int[] weightscopy = weights.clone();
      weightValue = weightValue *  weightDiminisher;
      weightscopy[i] = (int) (weightscopy[i] + weightValue);
      recursivelyClassify(T, classify, kanjis, weightscopy, bestcorrect, bestweights,
          weightValue, weightDiminisher);
    }
  }
```

6 Results

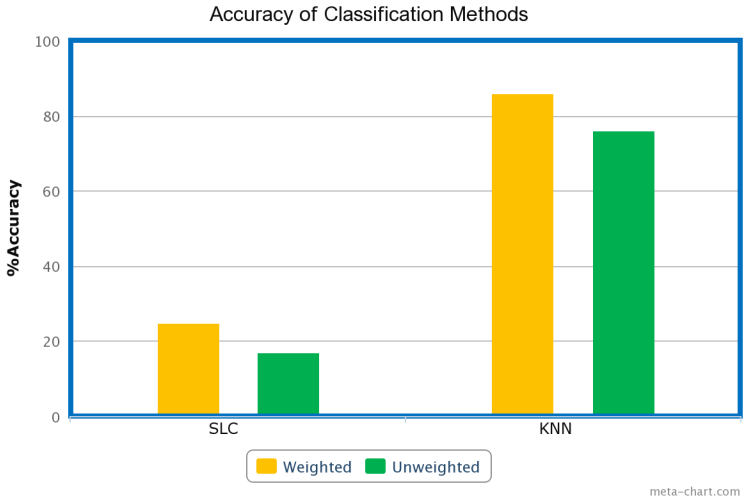

Figure 5: from left to right WCSL = 25%, CSL = 17%, WKNN = 86%, KNN = 76%

KNN strongly outperformed CSL by about 60% in both the 'weighted' and 'unweighted' cases. A hidden layer of weights improved the accuracy of both methods by about 10% (Fig. 5).

7 Discussion

CSL scored about the same as it does on the CASIA test in the original paper. However, KNN scored much higher. This discrepency could have something to do with handpicking good features for the data. Another explanation is that CSL needs a larger dataset to perform well. Finally, since the data has high variance it is likely there are errors in the descent which will result in inaccurate prediction.

The hidden layer experiment returned a rank for the importance of each feature. Features relating to the relationship between strokes and the whole kanji ranked first. Features relating to the relationship between strokes and other strokes ranked second. Features only relating to the strokes ranked last. In order of most important they are: midpoint distance from kanji center, angle between strokes, stroke length, stroke angle and number of strokes.

8 Conclusion

The data does not support my hypothesis that CSL and KNN would perform at about the same accuracy. KNN works very well, as expected. I expected CSL to work just as well since they performed about equally in the CSL paper (Chandrashekar, Ashok, and Richard Granger). However, CSL performed much worse than KNN on the dataset. I believe these low results for CSL may be due to choosing tree desecent as a classigication algorithm, which the CSL paper suggests is a bad algorithm to use on sparse data. I should run my classifier on the same dataset as in the CSL paper to get a better sense of whether or not I implemented the algorithm correctly. Also, I should run the experiment again with either a larger dataset or the KNN-leaves classification algorithm which might produce better results.

The data does support my hypothesis that both algorithms would be improved by a layer of hidden nodes affecting the feature weights. I ran both algorithms with a hidden layer of weights. Both jumped up their accuracy by 10% after the hidden layer. Humans are notoriously bad at picking features. The hidden weights layer performs the function of also ranking the features of the data. The features relating

to patterns, like the midpoint distance from the kanji center, were ranked more highly than features that described independent factors, like the length of each stroke. Thus, patterns appear to be more robust for both human and machine learning. The data shows that a mix of biological and engineering methods are best at classifying Japanese Characters.

Japanese characters are particularly interesting to classify since they have three layers of rigidly defined components: strokes, radicals and kanji. One interesting idea for a future classification study is to classify strokes into radicals and radicals into Kanji. This multi-layered structural classification could produce more accurate results from both KNN and CSL since splitting each Kanji into radicals will simultaneously decrease the number of types and increase the number of tokens in the data pool.

9 Sources

- Blondel, Mathieu, Kazuhiro Seki, and Kuniaki Uehara. "Unsupervised Learning of Stroke Tagger for Online Kanji Handwriting Recognition." Pattern Recognition (ICPR), 2010 20th International Conference on. IEEE, 2010.
- Chandrashekar, Ashok, and Richard Granger. "Derivation of a novel efficient supervised learning algorithm from cortical-subcortical loops." Frontiers in computational neuroscience 5 (2011).
- Duch, Wodzisaw, and Karol Grudzinski. "THE WEIGHTED kNN WITH SELECTION OF FEATURES AND ITS NEURAL REALIZATION."
- Tokuno, Junko, et al. "On-line Handwritten Character Recognition Selectively Employing Hierarchical Spatial Relationships among Subpatterns." Tenth International Workshop on Frontiers in Handwriting Recognition. 2006.
- Witten, Ian H., and Eibe Frank. "Weka." Machine Learning Algorithms in Java (2000): 265-320.