CSCI 737 — Pattern Recognition, Spring, 2016
Project #1

Michael Potter, Deepali Kamat

April. 2, 2016

1 Design

This document outlines pre-processing, feature extraction and non-baseline classifier tasks. Pre-processing discards redundant and irrelevant information from the data set.

1.1 Pre-processing Techniques

In order to eliminate the noise present in the data and to prepare it for feature extraction, the following pre-processing tasks were performed:

- Normalization: The data is scaled so that Y values fall within the range [0,1] while maintaining width to height aspect ratio.[1]
- Duplicate Point filtering: Duplicate points provide little, if any, information about the symbol being investigated, so only a single copy is kept.[2]
- Smoothing: Digital pens create data with uneven and discontinuous points. We employ averaging to counteract the jagged noise artifacts created by this technology.
- Re-sampling: Data point distributions are affected by the velocity at which symbols are drawn. In order to remove the effects of velocity from the stroke data, distance-based re-sampling is applied.[3]
- Second Normalization: The above pre-processing tasks can cause the y-axis bounds of the symbol to shift. These are normalized again to ensure consistency across symbol instances.

1.2 Features

Most features considered in this project were taken from the work of Davila et al[4]. The following features were extracted from the stroke data:

1.2.1 Global Features

1. Point count: How many points were required to form the symbol. Potentially useful for attempting to determine symbol size and complexity.
2. Normalized point count: Count divided by the number of strokes. Gives a complexity estimate for each component of a symbol.
3. Number of traces: The number of strokes required to create a symbol. Potentially useful for identifying symbols such as x or t.
4. Angular change: Tracks the extent to which a symbol curves. Useful for differentiating circles from lines.
5. Normalized angular change: Provides a more appropriate measure of curvature for symbols comprised of many strokes.
6. Line length: The line length of each trace in the symbol data. Useful for identifying long characters such as $\sqrt{...}$.
7. Normalized line length: The total symbol length was adjusted based on the number of strokes required to form it.

8. Number of sharp points: When there is a large change in the angle of writing, a sharp point is observed in the trace data. This tells us how may times a symbol might have experienced change in direction when being written.

9. Aspect ratio: The aspect ratio of the symbol is extracted by taking the ratio of the width to height of the bounding box in which the symbol resides.

10. X and Y axis mean values: The mean x and y coordinates are used to provide a measure of central tendency within a symbol. Symmetric symbols should have means around 0.5.

11. Covariance of x and y coordinates: The global covariance between the x and the y coordinates is computed to gain a general sense of directional trends in the data.

1.2.2 Crossing Features

Crossing features provide us with information regarding how many times the symbol traces intersect various lines. In certain symbols there are multiple intersections - providing detailed yet generalizable structural information. Binning the crossing profiles of collections of nearby lines helps to increase the robustness of the feature to handwriting variations.[4]

1.2.3 2D Fuzzy Histogram of Points

Dividing the symbol into regions and binning it can provide a spacial description of the data that is relatively invariant with writing style. Points are assigned to multiple bins based on their relative distances in order to further improve resistance to variable writing styles.

1.2.4 Fuzzy Histogram of Orientations

The orientation of points can provide regional information related to the symbol. The area of the symbol can be divided into regions and the angular relationship of the points with respect to the axis can be evaluated. This provides a style-invariant way to characterize how symbols curve through space.

2 Pre-processing and Features

2.1 Pre-processing

- Normalization
  As mentioned above, pre-processing our preprocessing begins with normalization. Normalization is performed by scaling the y coordinate of the points in the stroke such that they range between 0 and 1. The least and the greatest values among all values of y in a stroke are obtained. Each coordinate point in the stroke is then downshifted by this minimum value and divided by the maximum value to arrive at a normalized image.

- Duplicate Point filtering
  To remove the duplicate points in the data set, each point is compared with it's previous neighbor. If found to be equal, the point is deleted.

- Smoothing
  In smoothing, three consecutive data points are averaged to obtain a less jagged image. This is done to eliminate any jitter that may have been introduced when the digital data was recorded. The present, prior and next data points belonging to the x and y axis are each summed up and averaged. The new resulting stroke has lower jitter and a smoother curvature.

- Re-sampling
  To remove the effects of velocity on the data, the data is re-sampled using the following procedure[3]:

Algorithm 1: Re- Sampling via trace segmentation [2]

a. Input: S= $\{(x_0, y_0), \ldots (x_n, y_n)\}$, α
b. Output: $\hat{S}$= $\{(\hat{x}_0,\hat{y}_0), \ldots(\hat{x}_n,\hat{y}_n)\}$
c. L_0: Stroke length to be accumulated over all points
d. for i from I to n, do: //Compute accumulated distance
   i. $L_i = L_{i-1} + \sqrt{(xi - xi - 1)2 + (yi - yi - 1)2}$
   ii. m = = (L_{n-1}/α) //number of out points
   iii. $(x_0,y_0) = (\hat{x}_0,\hat{y}_0)$ // The first point is stored as is
   iv. j = 1
   v. for p from p to m-1, do
      while $L_j < (p.\ \alpha)$:
         j ++
         C = $(p.\ \alpha - L_{j-1})/(L_j - L_{j-1})$
         $\hat{x}_p = x_{j-1} + (x_j - x_{j-1}).C$
         $\hat{y}_p = y_{j-1} + (y_j - x_{j-1}).C$
      $(\hat{x}_{m-1},\hat{y}_{m-1}) = (x_{n-1},y_{n-1})$ // The last point is recorded as is
   vi. Return $\hat{S}$

2.2 Features

- Global Features:

  1. Point count: The number of points in a stroke is recorded.
  2. Normalized point count: The point count obtained is normalized using the total number of strokes in the symbol.
  3. Number of traces: The number of strokes present in the symbol is counted.
  4. Angular change: The angular change between points is obtained by taking the vector dot product between them. The angular change is the sum of the total change in slope angles between pairs of consecutive points for each stroke in the symbol.
  $$\theta = cos^{-1}\left(\frac{\vec{A} * \vec{B}}{\|\vec{A}\|\|\vec{B}\|}\right)$$
  5. Normalized angular change: The angular change divided by number of strokes comprising the symbol.
  6. Line length = Sum of the Euclidean distance traversed by each stroke in the symbol.
  7. Normalized line length: The line length for each symbol divided by total number of strokes in that symbol.
  8. Number of sharp points: The variation of angles helps determine if a point is a sharp point. This angle is calculated using the angular change formula. The angle is compared against threshold levels. The threshold was chosen based on subjective measure of sharpness.
  9. Aspect ratio: The maximum distance between points in the x direction divided by the maximum distance between points in the y direction.
  10. X and Y axis mean values: The mean values taken over all x and y coordinates.
  11. Covariance of x and y coordinates: The global covariance between the x and the y coordinates is computed based on all points in the symbol.
  $$cov(X,Y) = \sum_{i=1}^{N} \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}$$

- Crossing Features:
  The average horizontal and vertical crossing features of a symbol are computed by calculating how many times a symbol crosses over lines drawn out from the X and Y axis. Each axis is subdivided evenly into 5 regions. Each of these regions is then bisected by 10 lines used to monitor crossings. Once a line crossing has been identified, its coordinate is computed as:
  $$C = \frac{z_i - (z_i - z_{i-1}) * |z_i - threshold|}{|z_i - z_{i-1}|}$$
  where, z_i represents the point coordinate along each axis and C is the crossing coordinate being computed. The final crossing values are then normalized based on the total number of points in the symbol strokes.

- Fuzzy Histogram of Points:
  The symbol area is divided into a grid of 4 x 4 squares. This forms 25 corners with 16 bins. For each point in the symbol,

we calculate a membership with respect to the corners of the bin the point falls in. This membership is calculated using the formula:

$$m_p = \frac{w - |x_p - x_c|}{w} * \frac{h - |y_p - y_c|}{h}$$

where, m_p represents the membership value, (x_p, y_p) represent the points in the stroke, (x_c, y_c) represent the point coordinates of the corners, and w and h represent the width and the height of the bins respectively. This data is summed up for each corner and then normalized using the number of points in the stroke.

- Fuzzy Histogram of Orientations:
  The symbol area based on its minimum and maximum values is divided into a grid of 2x2 cells, creating 9 corner points. Each corner is assigned 4 bins. The bin values are used to represent angle membership. This is calculated by finding the angular change of a point with respect to the horizontal axis and then assigning weight to the two most similar bins. The bins each contain $\frac{pi}{4}$ radians, starting with a bin containing angles in the range $[\frac{-\pi}{8}, \frac{7\pi}{8}]$. The point membership values for each corner are calculated using the membership formula used for in histogram of points. The membership for each bin is obtained by summing up the membership of the center point of the segment; with respect to the corners, and adding the normalised value of the orientation angle membership.

3 Classifier

We chose to employ a random forest classifier for this project. The random forest was chosen for its ease of implementation / use, insensitivity to data normalization, and speed of training - advantages gained on top of a respectable accuracy rating. In particular, we employed the 'ExtraTreesClassifier' implementation provided by scikit-learn. This is a random forest implementation which allows the user to specify the number of features to be randomly selected from at each level of tree training. Based on the results of various tests described in Section 4, we decided to employ a random forest composed of 800 trees, each splitting based on gini impurity scores and considering every attribute at each level of split.

4 Results and Discussion

The first question we considered when thinking about features was whether they all ought to be normalized to the range [0,1] in order to allow for better comparisons of errors between features. This seemed like it should be especially important for the 1NN algorithm, though it is often advertised as being useful for classifiers in general. Surprisingly, we found that normalization of data was not helpful for our 1NN algorithm, which had a testing accuracy of 85.90% non-normalized but only 69.23% when normalized (when run on a sizable subset of our final feature set). This finding, along with the fact that decision-tree-based classifiers are not sensitive to normalization, lead us to omit the normalization process from our final design. We believe that in this case it happened that our features which have maximum values in excess of 1 (such as number of strokes) are also the most predictive, so leaving them non-normalized had the effect of lending more weight to more important features. Interestingly, when we expanded our feature set to its final form the 1NN testing accuracy actually dropped to 81.29%, indicating that features such as the orientation histogram and raw point count - while useful to more advanced classifiers - are actually detrimental to naive algorithms. This is probably due to the fact that, without normalization, the number of points in a given symbol will vary widely and is not as important as histogram features which are naturally normalized to a limited range. The 1NN system performance could be improved going forward by identifying the the most useful features and weighting them when computing error distance. It is not clear that this is worth pursuing, however, given that 1NN is a naive algorithm anyways and is unlikely to achieve results on par with our random forest classifiers.

We next sought to characterize change in performance in response to choice of features. A random forest algorithm, as described in Section 3, was employed to classify symbols. Our first benchmark was collected using global attributes, crossing features, and a fuzzy coordinate histogram. This resulted in a testing accuracy of 90.76%. One of the frequent errors encountered by this system was the confounding of the characters '1', ')', and ','. It was hoped that by adding an orientation histogram to the data we might better discriminate between these symbols. Unfortunately the addition of the orientation histogram did not dramatically improve performance on these characters, but did raise the overall testing accuracy to 91.75%. In a further effort to resolve the confusion, we attempted to create a feature that would provide a rough estimate of the size of a symbol: the number of points in the stroke before pre-processing. It was thought that symbols like commas would have only a small number of points and therefore be difficult to mistake for parenthesis and ones. Regrettably, this did not lead to a substantial reduction in the errors we were interested in, though it did cause the testing accuracy to further increase to 91.94%.

A list of the most common errors in this final feature set, based on the confhist utility, is given in Table 1. These errors are all easy to understand, especially given our inability to find a feature which strongly characterizes the relative size of each symbol.

Table 1: Common Errors for the Random Forest Classifier

Target Class	Total Errors	Most Frequent Erroneous Guess (Occurrence Count)
\times	150	'x' (143)
.	147	'-' (45)
1	138	'(' (33)
,	115	')' (41)
X	88	'x' (84)
z	79	'2' (46)
C	70	'c' (56)
|	52	'1' (46)
2	49	'y' (25)
g	49	'z' (23)

As shown in the table, the most common error made by the random forest was to call the times symbol ($\times$) an x. This is understandable since $\times$ and x look virtually identical. Next was a propensity to mistake the period symbol for subtraction. This was somewhat surprising given that one would expect periods to have far fewer points than subtraction symbols. It could be, however, that periods which do have several points are being stretched in our pre-processing step, making them appear more like a minus sign. Similar problems lead to the misclassification of commas, as well as the mistaking of capital letters for lower case letters. All of these problems might be alleviated by a feature capable of capturing the size of a symbol relative to the others within a file. This would prevent the re-scaling normalization from losing information about the intended symbol size. Unfortunately we have not yet been able to devise a feature that can provide this information without being thrown off by the relative arrangement of symbols in a formula. For now this remains an opportunity for future improvement.

The error profile for the nearest neighbor classifier was slightly different from those observed with the random forest. These errors are given in Table 2

Table 2: Common Errors for the 1NN Classifier

Target Class	Total Errors	Most Frequent Erroneous Guess (Occurrence Count)
1	353	'(' (76)
x	237	'\times' (75)
y	190	'g' (24)
,	115	')' (41)
2	180	'z' (37)
z	153	'2' (57)
n	145	'a' (44)
.	144	'-' (32)
4	132	'+' (16)
\times	118	'x' (82)

While many of the mistakes are the same between the random forest and 1NN classifier: $\times$ and x, 1 and '(', ',' and ')', the 1NN seems to have more symmetric errors ($\times$ confused as x and x confused as $\times$ both make the top 10). Also of interest is that the absolute error quantity for capital letters such as X and C is actually lower for the 1NN than it is for the Random Forest. This seems to imply that our features have the capacity to differentiate upper and lower case letters, but that some of this capacity is being traded away by the random forest in favor of better overall accuracy. In the future it may be interesting to attempt PCA on our feature space to see if the attributes responsible for capital letter sensitivity may be extracted into a more tree-friendly format.

In order to determine the best settings for use with our random forest, the effects of various parameters were investigated. First, the effect of tree count on classification accuracy was investigated, with a 500 tree forest providing the aforementioned 91.94% accuracy and an 800 tree forest providing a 92.03% accuracy. Given the marginal improvements offered by a heightened tree count, it seems unlikely that a further increase in trees would confer tangible benefits. Two different splitting

criteria were considered when building the trees: information gain (based on entropy) and gini impurity. The information gain method achieved an accuracy of 92.03%, whereas the gini method saw an accuracy of 92.06%. It therefore seems that the two methods are effectively equivalent, though without any hint of downsides the gini system was chosen for its marginally improved performance. Finally, the number of attributes chosen at random each level during tree construction was considered. The accuracy of 92.06% was achieved by selecting from amongst 50% of the features at each level. If instead fewer options are allowed (square root of the total number of features), the accuracy drops to 91.39%. Interestingly, this drop is in spite of a nearly 7GB increase in the file size of the saved forest (presumably a result of it driving the trees deeper in order to get a good split). It was therefore decided that considering more features per split is preferable. Based on that result, we investigated allowing the trees to consider all features at every split and found that this resulted in an accuracy of 92.16%. It therefore seems that for this problem, including feature drop out in the random forest is not advantageous.

Based on these experiments, we arrived at our final model: an 800 tree random forest which splits using the gini impurity metric and considers all features when performing a split. The effectiveness of this system is compared to that of our 1NN classifier in Table 3.

Table 3: Accuracy of Classifier models

Method	Training Accuracy	Testing Accuracy
1NN	99.99% (1 error)	81.29%
Random Forest	99.99% (7 errors)	92.16%

The first thing that stands out in Table 3 is that the 1NN training accuracy is not perfectly 100%. This is a result of single period being mistaken for a subtraction sign. While unlikely, this outcome is understandable since it is technically possible for multiple objects to have exactly the same feature representation, in which case the classifier will take the first perfect match it encounters. Of potential interest for those who maintain the lgeval tools, the evaluate function actually failed when trying to write the summary.txt file for these files, throwing the following error:

```
/usr/local/lgeval/bin/evaluate: line 170: /bin/cat: Argument list too long
Traceback (most recent call last):
  File "/usr/local/lgeval/src/sumMetric.py", line 456, in <module>
    main()
  File "/usr/local/lgeval/src/sumMetric.py", line 182, in main
    correctExps = int(allZeroCount["D_B"])
KeyError: 'D_B'
```

As one might expect, although the training error was lower for the 1NN classifier, the random forest performs much better on the test data set. While still not perfect, an accuracy of 92.16% is decent given the large number of classes to choose from, along with the high degree of variability in the writing styles of participants.

References

[1] L. Hu and R. Zanibbi. "HMM-Based Recognition of Online Handwritten Mathematical Symbols Using Segmental K-Means Initialization and a Modified Pen-Up/Down Feature". In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on*. Sept. 2011, pp. 457–462. DOI: 10.1109/ICDAR.2011.98.

[2] Y. Zhang, G. Shi, and J. Yang. "HMM-Based Online Recognition of Handwritten Chemical Symbols". In: *Document Analysis and Recognition, 2009. ICDAR '09. 10th International Conference on*. July 2009, pp. 1255–1259. DOI: 10.1109/ICDAR.2009.99.

[3] M. Pastor, A. Toselli, and E. Vidal. "Writing speed normalization for on-line handwritten text recognition". In: *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*. Aug. 2005, 1131–1135 Vol. 2. DOI: 10.1109/ICDAR.2005.257.

[4] K. Davila, S. Ludi, and R. Zanibbi. "Using Off-Line Features and Synthetic Data for On-Line Handwritten Math Symbol Recognition". In: *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*. Sept. 2014, pp. 323–328. DOI: 10.1109/ICFHR.2014.61.