CSCI 737

Pattern Recognition

Project 2

Handwriting Segmentation

Author 1 :
Justin Lad

Author 2:
Richard Riehm

Instructor:
Dr. Richard Zanibbi

April 30, 2019

1 Design Overview

Our segmentation program utilizes most of the classification program framework discussed in the previous project. Pre-processing remains almost identical (quick summary in next section). We add and change a few features to better describe symbols. Our segmentation approach includes a simple baseline where every stroke is considered a symbol, and a greedy segmenter that avoids a tremendous search space with fairly good performance. We also discuss progress towards the brute force approach.

2 Preprocessing And Features

We conduct very similar preprocessing to the previous project, repeated below. The note-able change here is how we handle preprocessing strokes for segmentation. Previously, preprocessing was applied to the entire stroke sequence in a file. This resulted in poor accuracy during segmentation, because the entire formula is scaled down into the normalized window between [0,1]. Consequently, symbols will be a scaled and shifted version than in training. To train, we use a python dictionary computed in the train/test split program, containing information about which strokes belong to each symbol. For testing, we preprocess groups of strokes in a hypothesis together.

Preprocessing involved a few steps. First, points are normalized into the domain and range of [0,1]. We implemented the algorithm from [**?**], replacing every pt x_i, y_i to a scaled version between 0 and 1. Unfortunately, the traces were upside down, so we flipped by multiplying points by negative one. While this probably doesn't impact the model's ability to learn the features, it felt natural to store them right-side-up, as it could come into play with segmentation. Then, points are smoothed by averaging the previous, current, and next points, inspired by reading [**?**] [**?**]. When plotting points, we observed that smoothing points twice resulted in more recognizable symbols, so we run the smoothing routine twice on the data. Finally, we used a de-duplication technique inspired by [**?**]. Instead of removing exact point matches (almost never occurred), we compute the L2 distance between points and remove anything within a threshold of 0.01. This feature improved accuracy by only about 1%, so this step could be left out if test speed is important.

3 Symbol Classifier

Our classifier is fairly similar to the previous project, with a few changes to improve performance. The classifier remains a random forest with 100 trees and a max depth of 25.

First, we introduce a new feature to compute angle of trace points, using the approach inspired by class discussion. The idea is to count occurrence of angles in 45 degree bins. We were originally going to calculate angle between points, done in [**?**], but then had the idea of computing angle between a line pointing due east, and the point. This way,

shapes like 0 would have a uniform-like number of angles in each bin, and shapes like 'x' would have most angles in 4 bins. To implement this, we find the centroid of each symbol hypothesis, and set it as the origin point. Then, we set the reference point at the same y value and x=1.1 (this ensure it is to the right of the largest x value). This creates a line that can be used as a reference for 0 degrees (due east). Then, the law of cosines calculates the angle from the origin point between the trace and reference points. If the y value of a trace point is lower than the origin point, the angle is greater than 180 degrees. Consequently, this angle is computed as (180 - calculated angle) + 180.

Second, we change starting and ending x,y coordinates to min and max x,y coordinate. We saw a 5% improvement in the previous project using these features. However, in the context of segmentation, these are very poor features to use. Not only can people write in different order, but when computing whether segments belong together, it caused order to matter. Instead, we compute the minimum and maximum x and y values and compute the distance between these points. Although the height to width ratio provides similar information, we found this improved classification, especially in the context of segmentation.

Training the classifier took 190 seconds 984 seconds., or about 3 minutes without the angle feature, and a surprising 9 minutes including the angle feature.

4 Segmentation

A segmentation algorithm is fairly daunting task considering the combinatoricly exponential search space of partitioning items. The Bell number describes the number of unique k partitions of length k list and explodes quite rapidly from 115,000 unique combinations at 10 and 4.2 million at twelve strokes. Therefore, we rely on a few key assumptions: limiting the number of strokes in a symbol to four, and only considering stroke groupings that are close together. Consequently, we add one additional feature for the segmentation algorithm: centroid locations of the strokes. These assumptions are used to construct a list of symbol hypotheses. We then implement a baseline and greedy segmentation approaches, and discuss progress on the brute force approach.

The first stage of the segmentation algorithm generates symbol hypotheses [**?**] [**?**]. Inspired by the MST idea in [**?**], we recognize that strokes close together in space are more likely to belong to the same symbol. Therefore, we compute the centroid of each stroke (average of max and min x,y values) and conduct a k nearest neighbor search. Because stroke neighbors are typically repeated (e.g. a plus sign), stroke groupings are sorted and only unique groupings are stored. Due to the constraint of maximum four strokes per symbol, the symbol hypotheses are a set of unique 1,2,3,4 length groupings based on nearest neighbors. Each stroke grouping is pre-processed and classified to produce a dictionary of stroke hypotheses. Although it takes more time to preprocess individual groupings in the segmentation hypothesis stage, preprocessing each stroke individually loses valuable information about the stroke, like the proper height to width ratio, variance, etc. No further classification is done once the symbol hypothesis are

generated; each stroke grouping serves as a lookup table when computing segmentation.

Our baseline approach simply segments each trace as a single symbol, predicting the most likely class for each individual stroke. While over-simplistic, it does a fairly good job because most symbols consist of one stroke.

Our main approach uses a greedy algorithm. It selects the stroke grouping with the highest probability, until every stroke has been selected exactly once. It outperformed the baseline classifier by 10%+ and took roughly [0.5] seconds on average to produce an lg file. The advantage to this approach is a massive reduction in the search space. Instead of computing a very very large number of permutations, it performs a worst case linear search among all the possible stroke hypothesis. Obviously, the downside is this is not a problem where the greedy property holds. For example, + is difficult to segment because individually, the vertical line could be a '1' or '-'. Contextual information and a global perspective is very beneficial in these situations, as shown in [**?**] [**?**]. However, if time is a concern, this is a reasonable approach. The algorithm is described in pseudo code below

```
def greedy_approach(computed_trace_groupings, num_traces):
    used_strokes = [False for number of traces]
    probabilities = [probability for each trace_group]
    segementation = []
    while not every stroke used:
        while not guess_used:
            best_hypothesis = argmax(probabilities)
            if no stroke used before:
                append trace_group to segementation list
                guess_used = True

            set probability of guess to 0 in probability array
    return segmentation
```

The brute force approach uses the recurrence relation of computing Bell's number in order to search through the entire search space of possible partitions. As mentioned before, this number explodes. With certain files containing up to 80+ traces, this approach becomes intractable. We propose this method to reduce the complexity of this brute force search. We first calculate a list of valid trace groupings through finding the probability of those traces both belonging to the same symbol. Any grouping of traces below a threshold of belonging together are ignored, and all above this threshold with be used in the proposed approach. Once we have calculated and obtained all groupings deemed possible, we use this list as input to our brute force approach as a validator, only pursuing likely paths in this recursive approach. The aspect of this approach that reduces the search space is limiting the maximum number of strokes per grouping. The pseudocode for this algorithm is:

```
Given P = [1....N] as identifiers to the traces
def Partition_P(P, V, k=4)
```

```
R = [] Resulting List of Partitions
if P > 1:
    Find all Pk of length k from P where Pk is in V
   loop from k to 1
        R += Pki + Partition(P - Pi, V, k)
    endloop
    R += Partition(P, V, k-1) if k-1 > 0

return R
```

The idea behind the above algorithm is that by limiting the recursion to 4 layers deep instead of N layers deep, we will be drastically reducing the computation time for a brute force approach to finding all the possible partitions using Bell's number. On top of that, we'd further limit the search space by only pursuing paths in which all of the partitions in it so far are valid.

After obtaining all the possible partitions via the method above, we'd still need to select the one partition as our choice for the given input. To do this we need to calculate the cost associated with each partition. We perform calculations of the probabilities of each individual group within each partition by determining the probability of the traces in that group belonging together. We then calculate our cost function: $\frac{1}{P}\sum_{i=1}^{P} 1 - p_i$, where P is the number of groups in the partition, and p_i is the probability of group i. This value is the "cost" of the partition, w. We would perform this cost function on all partitions and select the one that gave the lowest cost as our segmentation.

We weren't able to get this working quite yet. One of the biggest problems was how to avoid all the different permutations of traces within each of the groups, as well as duplicate partitions due to other paths. This problem would have caused this algorithm to not decrease the computation time as well as expected, which is why we decided to use the greedy aproach.

5 Results and Discussion

5.1 Train - Test Split

Splitting files for the 70/30 train-test split is a little more nuanced with many symbols per file. As we've explored in this course, class priors significantly impact the output of classifiers. Therefore, we approximately equalize class priors between the training and testing split. This is measured using KL-divergence, which computes the difference between two distributions. First, the true class priors are calculated by counting the number of occurrences of each symbol in each file. Once we've recorded the initial priors, we begin randomly moving files from the original set into the new testing set until there exists at least one of every symbol in it. This is due to the KL divergence requiring the number of classes between two distributions to be the same. Then, we look at a random file in the training set. We move that file into the testing set and recalculate both sets

	Recall	Precision	F-Measure
Baseline	67.35%	48.68%	56.51%
Greedy	75.31%	62.87%	68.53%

Table 1: Test Accuracy of OBJECTS

	Recall	Precision	F-Measure
Baseline	49.54	35.81	41.57
Greedy	58.16%	48.56%	52.93%

Table 2: Test Accuracy of OBJECTS + CLASSES

priors and then the KL divergence with them. If the KL divergence has increased after moving the file, the file is put back into the training set, and another file is randomly selected. If the KL divergence is lower than what it was before we moved that file, we accept this transfer and repeat until the size of the testing set is 30% of the size of the original data set.

5.2 Results

The greedy segmentation works better than anticipated, outperforming baseline by 12% on the test set f-measure. Segmentation time is a reasonable average of time of 0.55 seconds per file. The following tables describe the testing and training set accuracy. Figure 1 and 2 compare performance of approaches.

5.2.1 Testing Set

The confusion histogram tool (lg2dot) revealed some common errors with our program. Unfortunately, our classifier is not very good. Most errors involve single stroke symbols. The highest symbol error was '0', with 5,388 errors. Of those, 4,461 were single stroke errors (i.e. not a segmentation error). x, 2, +, -, y, a were the most common. This is puzzling because the angle feature was added to help differentiate these symbols. The next challenge were with '-', with 4,500 errors. Most common error, not surprisingly, are + and =. However, it did catch enough to improve f-measure by about 12%. Then, there's a large drop off to 1,000 errors for 1, ., and). The most common mistaken for 1 were x or +. This is a somewhat positive error, as it shows that the segmenter wasn't adverse to grouping strokes together. Lastly, the segmenter never guessed - for an = or + symbol. This indicates that the greedy approach only grouped traces together when it was very sure about doing so.

5.2.2 Training Set

The greedy segmentation approach performed about 20% better due to previously seeing samples. Similar to the test set, the classifier was the biggest problem. In particular, 0's

were very very bad. Of the 21,000 errors, 13,000 involved 0's. 9,000 of those were single stroke errors, with the top missed guess of -. 0 does not have much in common with -, so this indicates some sort of error with our features or issue with prior probabilities of the class. The next biggest error group was -, with 3,000. It confused - with = and + (makes sense), but also struggled with 7 and 5 (doesn't make sense). Same as before, with - as the ground truth, our segmented predicted = and + commonly, but = and + never predicted single stroke classes, indicating a downside with the greedy approach.

	Recall	Precision	F-Measure
Baseline	65.06%	45.51%	53.55%
Greedy	92.69%	84.86%	88.6%

Table 3: Train Accuracy of OBJECTS

	Recall	Precision	F-Measure
Baseline	52.03&	36.39&	42.83%
Greedy	72.04%	65.95%	68.86%

Table 4: Train Accuracy of OBJECTS + CLASSES

TrainSet Baseline	TrainSet Greedy	TestSet Baseline	TestSet Greedy
18.3 min	56.7 min	7.8 min	28.3 min

Table 5: Execution Time

6 Visualizing Results

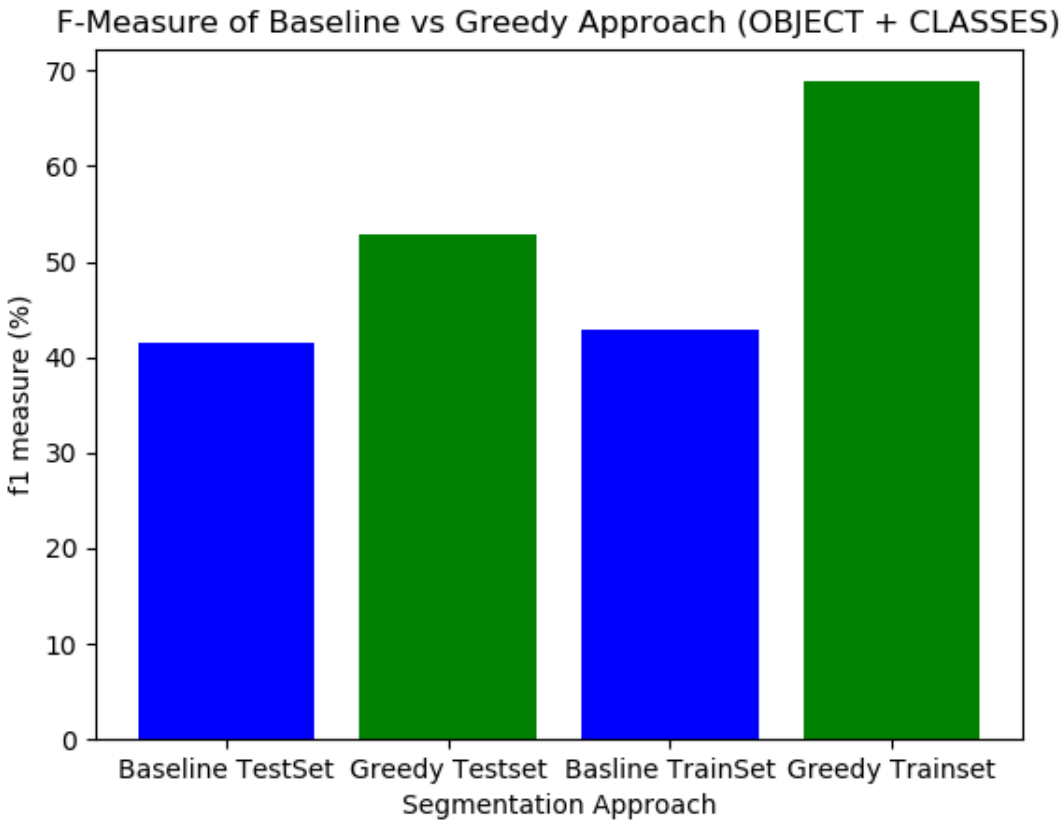

Figure 1: Comparison of performance measures of the correctly segmented and classified symbols

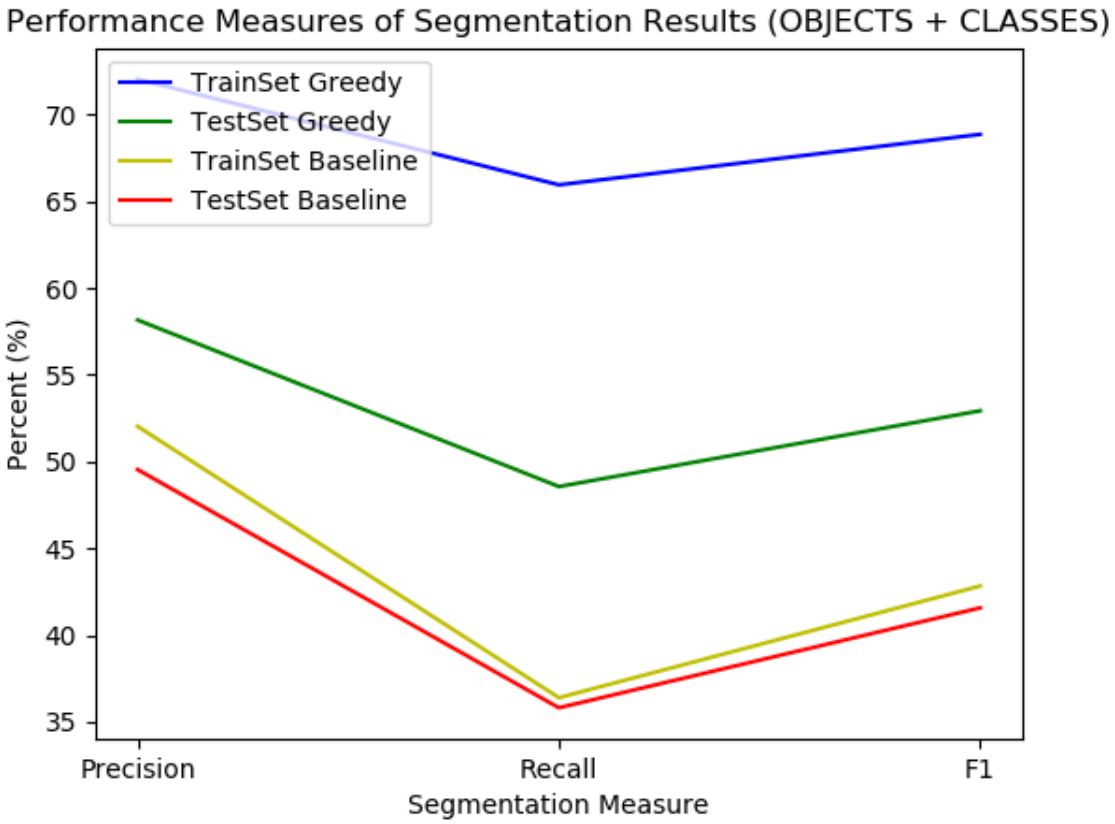

Figure 2: Comparison of F1 score of correctly segmented and classified symbols using baseline and greedy approach on train and test sets

7 Future Work

Overall, we recognize the greedy approach is not ideal, and see potential for implementing the brute force approach. It is a difficult algorithm to compute efficiently due to the constraints of only using a stroke once, and finding all the unique paths. It at least introduces some global context into the segmentation selection. We see a major issue of getting the brute force approach to execute in a reasonable amount of time on files with more than 20 strokes. Therefore, a hybrid approach could be implemented, where groups of 10 strokes are considered at a time. The downside is obviously strokes that occur between windows, but the trade off would be worth it in terms of time. We could also prune the symbol hypothesis generation by training the classifier with junk symbols, and only accept non-junk hypothesis. In general, dynamic programming is the correct approach, and it would be very useful to see an example of dynamic programming implementation for segmentation in this course. Finally, feature extraction could be much improved, as our segmentation would dramatically increase if was better at classifying single strokes.

This is probably the most important area, as it ripples into better segmentation.