CSCI 737

PATTERN RECOGNITION

PROJECT 1

Handwriting Classifier

Author 1 :
Justin LAD

Author 2:
Richard RIEHM

Instructor:
Dr. Richard ZANIBBI

April 4, 2019

1 Design Overview

The design of our handwriting classifier is fairly straightforward but provides a surprisingly high accuracy. Pre-processing involves size normalization, followed by smoothing, and de-duplication. Standard features are extracted, including number of symbols, (normalized) height, width, height/width ratio, distance between points, the number of points in the trace, starting x,y, ending x,y points, variance, covariance, and mean. Finally, we train a random forest as our non-baseline classifier, obtaining an accuracy of 83.4% on the true symbol + junk symbols test set.

2 Preprocessing And Features

Preprocessing and feature extraction is extremely important for classifier accuracy. This was the most daunting aspect of the project, but was much easier with groundwork paved by research papers. At the beginning of the project, we didn't fully appreciate that feature extraction *is* the task when building a classifier. Training models is somewhat trivial with packages like sklearn. In previous classes with toy classification problems, features are always given. It was fun to research and implement preprocessing and feature extraction techniques described below. Due to the relatively small number of features we had, dimensionality reduction was not used.

Preprocessing involved a few steps. First, points are normalized into the domain and range of [0,1]. We implemented the algorithm from [1], 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 [2] [**?**]. 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 [3]. 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.

Feature extraction relies on many fairly simple observations of math symbols. Certain features contain a lot of information. For example, the number of traces has a lot of information because it instantly eliminates the number of possible outcomes considerably. Most symbols are one trace, but two and three trace symbols are much more likely to make the correct guess due the small number of different classes that contain them. Similarly, the aspect ratio of a symbol is an important feature, as it can clearly be a distinguishing trait between symbols such as "1" and "O" or "=". The aspect ratio is calculated by dividing the height of the symbol by the width: Aspect ratio $= \frac{(y_{max} - y_m in)}{(x_{max} - x_m in)}$

The total distance drawn can be an important feature, as it can vary drastically between some classes. Due to the normalization in the preprocessing, the distance drawn within each class will be very similar. It is important to have features that distinguish classes from each other while staying relatively the same within each class. In order to calculate the total distance drawn for a symbol, we calculate the euclidean distance between each point for all traces.

We used the starting and ending x and y positions as four different features for classifying symbol. The x and y positions for both start and end coordinates are kept separate as to distinguish classes that have similar x position yet varying y positions. Although there is a potential for slightly more within class variance for these features(e.g. an "O" could potentially start at any point), they tended to significantly improve the accuracy of our classifiers.

Following from the four features mentioned above, we determined there could be use for the euclidian distance between the starting and ending point of a symbol. Adding this feature could alleviate some of the problems caused by the above features, as described. For example, while classifying an "O" could be problematic by the features above, it can be resolved by this new feature. While the start and end points could theoretically be anywhere on the line, the distance between the two points will be close to 0 for every "O".

Another feature that helps distinguish symbols is the number of points that intersect a vertical or horizontal line drawn through the center of the canvas. Since the symbols are comprised of points, one of these points may not lie on the intersecting line, but when drawn, it appears as though it should. In order to calculate how many times an intersecting line crosses points in a symbol, we use adjacent points within each trace as their own line segment, and then check if that line segment intersects the intersecting line. We then add the total number of intersections for each. This is directly copied from [4].

```
def ccw(A,B,C):
    return (C.y–A.y)*(B.x–A.x) &gt; (B.y–A.y)*(C.x–A.x)

def intersect(A,B,C,D):
    return ccw(A,C,D) != ccw(B,C,D) and ccw(A,B,C) != ccw(A,B,D)
```

We used the means for both the x and y values of each symbol as two different features. While these features may not seem like much, they can still help with the differentiation of classes (e.g. a “6” will have a much lower y mean value and a “9”, and a “(” will have a lower x mean value than a “)”).

Lastly, we used the var(x), cov(x,y), and var(y)as three separate features [5]. Similar to aspect ratio, the variance and covariance describe the spread of coordinates for each symbol.

3 Classifier

To test feature extraction, the sklearn KDTree is used as a baseline classifier. This is a simple but surprisingly effective model for mapping features into a k-dimensional space, and getting the closest 10 symbols on a test symbol. The only parameter to tune the tree is the distance metric for queries. We used the default Euclidean distance, as it made the most intuitive sense and provided similar accuracy to the random forest.

The random forest classifier was implemented with the sklearn RandomForestClassifier. Random forest is fast to train and test, and seemed to be a natural choice based on the structure of the feature vector. For example, features like number of traces, symbol width and height can provide a lot of information, especially ensembled together. In this implementation, randomness is injected in two ways. Each tree is constructed by randomly sampling training data (selected by the *bootstrap* boolean) and randomly selecting which features to split on (selected by $max\_features$). We had 15 features, so 4 features were used in each tree using the default $\sqrt{n\_features}$. We tried using all features, but this increased the pickle size and not surprisingly reduced performance by about 0.5% due to overfitting.

A few key parameters tune the shape and structure of the random forest ensemble. Parameters were tested by trial and error to see which features had the largest impact on performance. Random seed was used to ensure fairness when tuning parameters. Tuning these parameters provided an roughly a 5% improvement from the default random forest. This underscores the importance of feature extraction; a classifier is only as good as the data it trains on.

As long as key parameters are in a reasonable ball park, $n\_estimators \approx$ 50+ and $max\_depth \approx 20$, the random forest will perform within roughly a 1-2% accuracy. Increasing the ensemble size only improved performance, but reached a point of diminishing returns at approximately 100. Increasing to 200 only improved performance by about 0.2% and increased the pickle size from about 500MB to 2.5GB. Similarly, using a max depth of 5 decimated results, reducing accuracy by about 20 %. A max depth of 20 to 25 seemed to be about the sweet spot for best performance. Using a max depth of 50 reduced performance by about 0.2% and also increased size of the pickle. Finally, information gain was used over gini criterion because it improved performance by about 0.4%. The rest of the parameters were set to the default, as they made a negligible impact on performance.

4 Results and Discussion

The random forest classifier outperformed the KD-Tree by 5-10% on the various test sets. It was surprising to see how the simple structure of a KD-Tree can perform well if the features are well defined. For the following results, we train the KDTree and random forest on the true symbol + junk training set and test on true symbols and true symbols + junk. Results discussed in this section use the random forest with parameters discussed in the previous section. Please note that our submitted random forest uses a forest with 25 ensemble 25 depth tree to cut down on the pickle size.

The junk class was the number one incorrect guess for all classifiers and test sets by a very large margin. guess for each classifier and test set is the junk class. This could be due to much larger number of junk symbols in training compared to the number of real symbols. Additionally, junk symbols do have features in common with real symbols, so it is reasonable to mis-predict. Therefore, the discussion of confusion matrices will ignore junk symbols.

In general, the classifiers were not great at differentiating symbols that have a similar shape like 'z' and '2.'

4.1 True Symbols Only

	KD-Tree	Random Forest
w junk	66.40%	68.71%
w/o junk	72.46%	83.99%

4.1.1 KD-Tree Classifier

A very common error was misclassifying ‘z’ and ‘2’. Interestingly, it confused ‘y’ with ‘3’. This could be due to the fact the the start and end points are in roughly the same location for each. It also confused ‘x’ and ‘times,’ which makes sense because they have very similar shape.

4.1.2 Random Forest Classifier

This classifier also mixed up Z’s and 2’s most often. It also confused x and X, which could be explained our preprocessing step of normaliziation. Finally, it confused ‘1’ with ‘(’ and ‘)’ often, which is explained by having many similar features.

4.2 True Symbols + Junk

	KD-Tree	Random Forest
w junk	76.56%	82.07%
w/o junk	71.91%	83.38%

4.2.1 KD-Tree Classifier

The KD tree improved on confusing ‘1’. Similarly, ‘z’ and ‘2’ is improved. We think this is likely due to previously junk predictions becoming correct, now that junk symbols are in the test set.

4.2.2 Random Forest Classifier

With both true and junk symbols, the random forest still often misclassified ‘z’ and ‘2’, but at a roughly 30% less rate. Again, it mixed up ‘1’ with ‘(’ and ‘)’, but at again at a lower rate. Again, this is likely due to previously junk predictions becoming correct, now that junk symbols are in the test set.

4.3 Future Improvement

Resampling would be useful for samples where many points occur close together, due to the speed at which the symbol was drawn. This skews the mean and variance of the whole points list of that symbol. By resamlping, it would improve the accuracy of some of our features.

We may also add additional features or consider a different approach to normalization that allows catching nuances of symbols that have very similar features like ‘x’ and ‘X.’

It would also be interesting to see how a state of the art model like ANNs could help improve classification rate. We suspect it could achieve an additional few percent, but nothing significant.

References

[1] V. Deepu, S. Madhvanath, and A. G. Ramakrishnan, “Principal component analysis for online handwritten character recognition,” in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 2, Aug 2004, pp. 327–330 Vol.2.

[2] M. Pastor, A. Toselli, and E. Vidal, “Writing speed normalization for on-line handwritten text recognition,” in *Eighth International Conference on Document Analysis and Recognition (ICDAR’05)*, Aug 2005, pp. 1131–1135 Vol. 2.

[3] B. Q. Huang, Y. B. Zhang, and M. T. Kechadi, “Preprocessing techniques for online handwriting recognition,” in *Seventh International Conference on Intelligent Systems Design and Applications (ISDA 2007)*, Oct 2007, pp. 793–800.

[4] B. Boe, in *”Line Segment Intersection Algorithm”*, 2006, https://bryceboe.com/2006/10/23/line-segment-intersection-algorithm/.

[5] K. Davila, S. Ludi, and R. Zanibbi, “Using off-line features and synthetic data for on-line handwritten math symbol recognition,” in *2014 14th International Conference on Frontiers in Handwriting Recognition*, Sep. 2014, pp. 323–328.