CSCI 737 Pattern Recognition Project 1

Satwik Mishra Nikunj Kotecha

1 Design

In this project we were provided with inkml files (CHROME 2019) that contain information about the symbols encoded in trace points. Each trace is either a 2D array, which determines the x, y points or a 3d array, where the third axis determines time. In our implementation we have ignored the time variable if present. We begin with extracting this information from every inkml file along with the unique identifier for each file. Preprocessing steps are carried out on the 2D array of traces. It involves scaling the trace points in the range of $[0, 200]$ for each file in order to make the proportion between trace similar. Next, a total of 63 features were extracted for this problem. In this eight are global features, which describe the symbol as a whole. Moreover, there are a total of 30 crossing features, which are extracted from the images that are generated by plotting the traces using matplotlib python library [**?**]. These features describe the shape of the symbol with respect to specific regions horizontal and vertical direction. There are another 25 features extracted, using the concept of 2D fuzzy histograms, which captures the distribution of the trace points in the scaled region with respect to fixed positions(corner points) in that region [**?**]. More on these features are explained in the following sections. These features are then dumped into csv files, which are then split into training and testing set in the ratio of 70 to 30. The 70% is then used to train our classifiers, after which the trained model is used to make predictions on the 30% of the data set. Combining these features together, two models were trained, 1-NN(KD-Tree) and Support Vector Machines.

2 Pre-Processing and Features

2.1 Pre-Processing

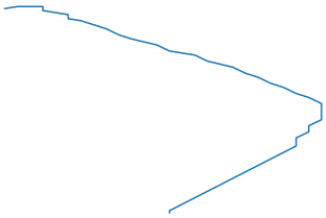
2.1.1 Interpolation

The idea is to have a set of points in our stroke such that are spaced apart by some constant distance. This distance d, is basically the average of the lengths of all the line segments in a stroke. To do so, we pick a point p_i, and look for the next point p_{i+n} such that the euclidean distance between those points is more than d, d is given by equation. Now that we compute the *(new_x, new_y)* i.e. d distance away from p_{i+n} by using equations. Eventually we will have a list of points in the stroke such that are spaced apart almost equally. This helps to reduce the variations caused due to different writing styles and stroke speeds.

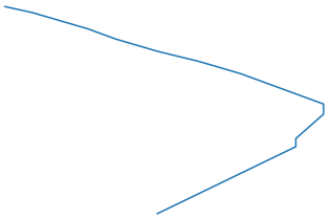
2.1.2 Sharp Points

2.1.3 Hooks Removal

Hooks are basically the those parts in a stroke which are a result of pen down and pen up actions or random hand movements, and usually occur at the beginning or end of a stroke. These are characterized by three properties the position, the size (small in length), and the change in angles [2]. After the computation of the sharp points, we check if any of those sharp points is a hook. Only if there are more than two sharp points further processing is required. Form line segments between the first two and last two sharp points. Lets us name the *seg_b* and *seg_e* respectively. Compute the slopes β_1 and β_2 of both the segments. There are two threshold values, one for angle and one for the length. We compare the length of the line segments and the angles between and with the threshold angle value (set at 90 degrees).



(a) data without any interpolation, and we can see hooks at the end



(b) After interpolation, and removal of hooks

2.1.4 Duplicate Point Filtering

We filter out the unique trace points getting rid of the duplicates. As those duplicate points don't add any extra information.

2.1.5 Smoothing

Smoothing helps to reduce the finger jitters caused while writing. We simply moved an averaging window over the list of trace points, where, for every point we replace it with the average of the current previous and next point.

2.2 Normalization

Normalization helps to reduce the variations and noise brought about due to different writing speeds, and the different ranges in which the coordinates lie. We normalize the y coordinates in the range $[0, 1]$ and then scale x such that the aspect ratio is maintained. Moreover while normalizing its is **important to note**, that normalization is done over all the strokes in the expression and not just one isolated stroke considered at a time. If you normalize a single trace, considering only its coordinates at a time, then all the (x,y) will be condensed to a single bounding box, loosing all the structural information as shown in Figure . Figure , shows the normalization of a stroke after considering all the stroke coordinates.

2.3 Feature Selection

In order to classify math symbols, unique features need to be generated to identify and distinguish one symbol from the other. As mentioned in Section 1, we have broken down the feature space into three categories for a total of 63 features, global features (8) [**?**], 2D Fuzzy histograms of corner points (25) [**?**] and Crossing features (30) [**?**]. Hence, a total of 63 features.

2.3.1 Global features

The 8 features extracted help to get a sense of the overall image. These features include the number of strokes (the total number of points present in traces) (1), number of traces (1), aspect ratio of every symbol (1), mean aspect ratio in x, y (from images) (2), mean of all points in x and y direction for a given trace (2) and lastly the covariance between the x and y of the traces (1). These features each represent the symbol individually and provides good features to distinguish one from another.

2.3.2 Crossing features

This method helps us quantify the shape of every symbol as a vector of 30 features. These features are extracted from the image of the symbol after plotting all the traces. For all image processing we used the Opencv library for python [**?**]. In order to maintain uniformity, these images are resized to 200x200 size. We decided 200x200 as for lower dimensions it was leading to information loss. To find the features, an image is divided into 5 different regions, in the horizontal and vertical direction. In each of these regions, 9 lines are drawn at equidistant space as shown in Figure 1, as suggested in the paper [**?**]. The number of intersecting points are averaged for each of these regions. Apart from these, the first and last intersecting points are also averaged for every region in each direction. Hence, a total 10 features for 5 regions, in each direction. Another 20 features for first and last intersection over 5 regions in each direction. Thus a total of 30 features were extracted.

cross_features.png

Figure 2: Crossing Features, where V1,...V5 correspond to the 5 vertical regions, H1...H5 correspond to the 5 horizontal regions.

2.3.3 2D Fuzzy Histograms

The traces, after being normalized and scaled, are divided into to Grids of 4-by-4 cells. Now for every point in the trace, we compute the association with all the corner points of the cell in which it lies as shown Figure 2. So for a trace point $T = (x_t, y_t)$ and a corner point $C = (x_c, y_c)$, the membership value of T is given as $m_t = \frac{w- \mod (x_t - x_c)}{w} * \frac{h- \mod (x_t - x_c)}{h}$ where w, h are the height and the width of that cell [1]. On dividing it into grid of 4-by-4 cells, we get 25 corner points. Then the membership values are computed for a trace point corresponding to all the 4 corner points in the same cell as that of the trace point. Finally, for each corner point, the average of the membership value is computed and returned as a vector of size 25. This is a great feature as it captures the

density of points in the scaled space, as we are basically computing that how many points and the degree of association those points in the traces have with a fixed corner point in the space. This feature significantly improved are classification accuracy.

Figure 3: 4-by-4 cell Grid layout for computing fuzzy histograms. For the green points, we will compute the membership value with the solid red corner points

2.4 Standard Scalar

Final preprocessing step is applying scalar standardization to the extracted feature set. For a feature point x, the standard score is computed as $\frac{x-\mu}{\sigma}$, where μ and σ is mean and the standard deviation of the feature set [**?**, **?**]. This is an important step, as we use SVM with RBF kernel, and it is important for the data to somewhat fit the distribution as that of a normal distribution. This also prevents any particular feature to dominate the decision making just because it has a higher value than other feature types.

3 Classifiers

3.1 K-Nearest Neighbor classifier using K-Dimensional-Tree algorithm

Parameter	Value
n_neighbors	1
leaf_size	30
algorithm	kd_tree
p	1
n_jobs	-1

Table 1: Classifier Final Parameters for 1-NN (KD-Tree)

We used the implementation of K-NN with KD-tree algorithm provided by the scikit-learn library for python [**?**]. Since we are working with 1-NN, the decision boundary will have high variance and low bias. The use of kd-tree helps in reducing the time for generalization as it reduces the search space by splitting it along the data axes at the mid points [**?**], this reduces the number of computations to get the distance from all the neighbors, as we no longer need to compare with every neighbor, but only the neighbors in the same region/space as that of the unknown point, after the space has been split into regions using the KD-Tree algorithm. Scikit-learn provides few parameters like leaf-size and the metric of measuring distance. We tried changing the leaf size and the distance metric parameter to see if there is any change in the classification and performance

of the classifier. We observed that there was no change in the accuracy achieved on varying the leaf size. However for the given data we got better results on using the Manhattan distance (p=1) as compared to euclidean distance(p=2) as our metric as shown in Table 5 in results section. The parameters selected for 1-NN Kdtree classifier are shown in Table 1. Also setting the *n_jobs=-1*, it uses all the available cores to speed up the training process.

3.2 Support Vector Machines

Parameter	Value
C	100
gamma	auto
class_weight	balanced
kernel	rbf

Table 2: Classifier Final Parameters for SVM (RBF Kernel)

On including the junk files with the valid set, we realized that it will cause a huge imbalance in the data set, so we wanted to see if we could tune the class weights somehow. Secondly, we also wanted to control how much influence a point can have on the decision making depending upon how far or close it is from the boundary. Finally, regularization, which helps us prevent overfitting. SVM classifier (one vs one for multi class classification) implementation by scikit-learn [**?**, **?**] was the most suitable option as it lets us tune these parameters by using the "class_weight", "Gamma" and "C" parameters of the classifier. We set gamma value to 'auto', which as per the sklearn documentation is computed by *1/number of features*. For lower values of gamma, points far from the boundary will impact the decision making, for higher values the closer points will be more influential. We set it to auto, as our number of classes are 101 (without junk), lower value will ensure that there is less chance of getting a very intricate decision boundary, that is affected only by the support vectors. With an intuition that this classification problem is not linearly separable we didn't try the linear kernel nor the poly kernel as for polynomial there is a need to find a suitable degree as well. So we went ahead with the Gaussian kernel, and we had accordingly prepared our data by processing it by standard scalar transformation. The training and testing split maintains the prior probability of the classes in both the sets, however we are working with classes with drastic differences in the frequency of occurrence within the train sample itself. The class_weight, which basically tells us the factor by which we penalize for a miss classification for a particular class, is set to 1 by default. However to counter the class imbalance, we used class_weight= "balanced". This sets the class weights inversely proportional to the frequency of occurrence that is given by total samples/(no of classes * frequency of each class), where frequency is computed from the ground truth provided. Hence for an underrepresented class, the class weights will be higher and more penalty for a miss classification for that class. The parameters selected for SVM (RBF Kernel classifier) are shown in Table 2.

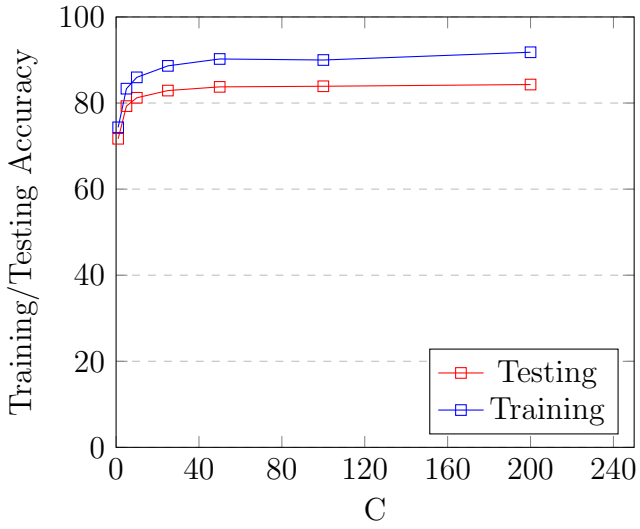


Figure 4: Comparison of traning and Testing accuracy on varying C parameter of SVM classifier

We tried to observe the change in the decision making on changing the value of C, keeping other parameters constant. As we can see from the figure, the training and testing accuracy both go up as we increase C, the greater the value, the more complex our model is. We also observed that, overfitting also increases as we keep increasing C. At C=100, there is a slight dip in the gap between training and testing accuracies, and then the gap increases again. So we finalized C at 100.

4 Results

Data trained On	Classifiers	Top-1 (W/o Junk, Merged Classes)	Top-1 (With Junk)
Valid	1-NN KD-Tree (Manhattan)	87.65	-
	SVM RBF Kernel	91.45	
Valid + Junk	1-NN KD-Tree (Manhattan)	79.7	80.93
	SVM RBF Kernel	88.13	85.23

Table 3: Comparison of accuracy for 1-NN + KD-Tree and SVM based on results from evalSymbolIsole.py for training splits

Data trained On	Classifiers	Top-1 (W/o Junk, Merged Classes)	Top-1 (With Junk)
Valid + Junk	1-NN KD-Tree (Manhattan)	66.87	71.06
	SVM RBF Kernel	75.70	78.33

Table 4: Comparison of accuracy for 1-NN + KD-Tree and SVM on **Unseen data**

Data	Classifiers	Top-1
Valid	1-NN KD-Tree(Euclidean)	86
	1-NN KD-Tree (Manhattan)	86.43
Valid + Junk	1-NN KD-Tree (Euclidean)	78.86
	1-NN KD-Tree (Manhattan)	80.27

Table 5: Comparison of accuracy for 1-NN + KD-Tree: Euclidean vs. Manhattan distance metric, on the 70-30% split data for valid and valid+junk

We trained and tested are classifiers on the CHROME 2019 data set. The training and junk set has a total of 85802, 74284 inkml files respectively. Table 3 shows the results of classification on 70/30 split on the valid, and valid+junk data respectively. We achieved an accuracy of 88.13% for classification of valid and junk points using SVM with RBF kernel, and it performs better than Kd-Tree in both the cases. Possible reason would be that the knn with kdtree is overfitting as k=1. One of the ways we can prevent overfitting is by increasing the size of k from one to some other greater number which would make our model less complex. As we can see in Table 3, SVM performs better at classifying valid and junk than 1-NN(kdtree). Secondly we were able to factor in the class imbalance by using the class_weight parameter which is not available in the KNN algorithm. Although we are using the *class weight* parameter to counter the imbalance in the data set, while training on valid+junk data, the junk out numbers all the other classes by a huge margin, because of which the most frequent confusion made in both is basically missclassifying valid symbols as Junk class. However SVM performs better than the KNN (Kd-tree). Other confusions made: "1/C/r/l/|" are classified as "(" or ")" for both KDtree and SVM, however the number of these missclassifications are less for SVM. Hence for very similar looking symbols, it gets difficult for our mathematical models to make decisions. For the unseen data as well SVM performs better, as shown in Table 4. The reason for this would again be the class imbalance that is brought about by the Junk files, because of which it is easier for a classifier to assign more classes to junk, in order to increase the classification accuracy. But we do tackle this in our svm classifier by the *class_weight* parameter but not in our baseline 1-NN. Another reason would also be that for the SVM classifier we trained on the entire data set, but for kd-tree we trained only on the 70% of the training data split as per the requirements of the project.
When trained on the 70/30 split, the SVM classifier missclassifies 6.03% of the valid symbols as Junk, as compared to 11.43% for 1-NN. However both perform similar in terms of miss-classifying junk as valid symbols i.e. 18.11% and 17.64% for 1-NN and SVM respectively. We can improve the accuracy by carrying out more preprocessing like removing duplicate points from the trace, and smoothening the traces. Computing features like sharp points and taking in the angular change as a feature, oversampling or under-sampling the data to balance out the class imbalance and lastly, instead of using only one classifier, using an ensemble or stacking of multiple classifiers might give a boost to classification rate. Currently we are extracting features by processing 200x200 size images, which increases our feature extraction time. We can reduce this time by incorporating a multi-threaded feature extraction process and also by finding a more suitable size for an image, that provides us with good features without the loss of important information.

References