Classification of Online Handwritten Mathematical Symbols (Project 1)

Report By: Sanyukta Sanjay Kate (ssk8153), Pratik Sanjay Bongale (psb4346)

Design

Our classification pipeline includes the following modules: parsing raw data(.inkml files), perform preprocessing, extract features and train the classifier, test and evaluate classifier.

The data in the inkml format was parsed using BeautifulSoup, we extracted the UI tag and trace information. We followed the procedure for preprocessing as described in [1]. Four preprocessing steps, namely, removal of duplicate points, smoothing/averaging, size normalization and resampling of points(speed normalization) were performed. We obtain fixed set of points describing every symbol from after resampling, we generate online features for every point in the symbol as described by Hu & Zannibi (2011) [1]. The features cosine of the slope, sine of the curvature and the normalized y coordinate form a three dimensional feature vector for every point in the symbol. We used Kd-Tree and Random Forest classifiers for the classification task.

Preprocessing and Feature Extraction

We have implemented four steps as described in [1] for preprocessing the trace information obtained from each inkml file.

a) Removal of Duplicate Points: If a point was repeated in the trace, it was removed from the trace. This is necessary in preprocessing as repeated or overlapping points do not convey any new information and hence, by removing the duplicate points we reduce the computations done by the classifier.
b) Smoothing: smoothen symbol trajectory by taking a running average of the points along the trajectory. This includes replacing the x, y values of current point with average of previous point, current point and the next point. Smoothing reduces jitter which is very common when writing with a digital pen [1].
c) Size Normalization: normalizing y-coordinates of each point to be between 0 and 1, modifying x coordinates maintaining the aspect ratio of the symbol. Every point's y coordinate is subtracted with minimum y coordinate value(among all strokes) and divided by the range of the y coordinates (considering all strokes of the symbol). The range of the coordinate is found by taking the difference between the maximum and minimum value of the y coordinate. When we change the y coordinate of a point, the x coordinate of that point needs to be scaled relative to the change in y and aspect ratio of symbol. We

normalize x coordinates by dividing them with the range of y. The aspect ratio is calculated by $1/(y_{max} - y_{min})$, so x_i * aspect_ratio provides the scaled values of x. While normalizing the symbols, we came across symbols which had only one point or points which have the same y coordinates, for example, the 'dot' or hyphen '-' symbol. Due to this, the maximum and minimum value of y was the same which lead to 'divide by 0 error'. We handled this issue by incrementing the maximum value of y by 100 and by decreasing the minimum value of y by 100. We used 100 because it was easier for calculations. Any other number can also be used instead of 100 (except 0).

d) Resampling: The data points in each symbol are equidistant in time but not in space. To make the points equidistant in space we resampled each symbol such that the total number of points over all traces is 30. As every symbol can be made of multiple strokes/traces, we calculated the number of points which every stroke should have. Hence, we divided 30 by the number of strokes in that symbol. Remainder points were adjusted among the strokes by adding 1 extra point to each stroke until we reach exact 30. We fit a curve for each stroke using the 'splrep' function from the scikit–learn library. We then find/sample the x,y coordinates of this curve at equidistant points in space. The curve fit over each stroke is a B-Spline curve of degree k=3. We got a few issues while resampling the symbols, for example, the splprep function worked only if there were a minimum of 4 points in the stroke. Hence, when we had a trace with less than 4 points we got a type error of 'm>k must hold'. We fixed this issue by generating additional points by interpolation of existing points. If there was just 1 point in a stroke, we created another point by incrementing the existing point's x coordinate by 0.01 and then interpolating these points using the linspace function from the numpy library. If there were two or three points in the stroke, then we would just add 4 points between the two consecutive points using the linspace function. The other issue we had while resampling the trace points was that the splprep function did not take consecutive duplicate points. Initially, if we just had 1 point in the trace then we would just create 4 points manually by incrementing and decrementing the x and y coordinates and if any of the values went below 0, we would take the absolute value of it. For few samples, taking the absolute values created duplicates and hence, we got a system error. To avoid this, we had to change our idea of manually creating 4 points and hence, used the above idea of interpolating between two points. This removed the issue of duplication and the trace points less than 4 and each symbol in the dataset had 30 points at equidistant space.

After preprocessing, we extracted features for each point in the symbol as per [1]. We created a 3-dimensional feature vector of each point in the symbol. Hence, each symbol would have 30 * 3 = 90 features. Hu & Zannibi (2011) used five online features which were the cosine of the slope, sine of the curvature, the normalized-y coordinate, pen up pen down and normalized distance to stroke edge (NDTSE). From these above five features, we decided to use 3 features, which was the cosine of the slope, sine of the curvature and the normalized y-coordinate. We did not use the pen up and pen down feature as we had considered all the points drawn to be pen down. If we had to consider the pen up feature, we would have to connect the last point of one trace with the starting point of another trace. Hence, we decided to leave out this feature. The NDTSE feature was a modification of pen up/down feature.

Features:

a) Cosine of a slope: Cosine of a slope is a part of vicinity slope as described in [1]. If we have a current point as $P(x,y)$, then we take the angle formed by joining $P(x-2,y-2)$ with $P(x+2,y+2)$ and horizontal line passing through $P(x-2,y-2)$. The cosine of this angle is taken, thereby giving out first feature for the point $P(x,y)$. We did not calculate the cosine of the slope for the first two points and the last two points of every trace in a symbol, as to calculate the cosine of the slope we require $P(x-2,y-2)$ and $P(x+2,y+2)$ for every point. We referred to the diagram described in [1] to get a better understanding of the cosine of the slope.

b) Sine of a Curvature: The sine of curvature is a part of curvature as described in [1] which describes the sine and cosine of the angle taken between the lines joining the current point $P(x,y)$ with $P(x-2,y-2)$ and the line joining the point $P(x,y)$ with $P(x+2,y+2)$. [1] We did not calculate the sine of curvature for the last two points and the first two points of each trace of a symbol as $P(x-2,y-2)$ and $P(x-2,y-2)$ is required for every point in the trace.

c) Normalized y-coordinate: the third feature vector is the normalized y coordinate which we had calculated while normalizing the symbol. We just add the current point's y coordinate $P(y)$ into the feature vector as it is the normalized y coordinate.

Classifiers :

The next part of our project was selecting the classifiers. We trained the existing classifiers of scikit-learn and stored the models as pickle objects. The first classifier that we implemented using scikit- learn was the ‘Kd-Tree’ Classifier and the second classifier which we chose was the ‘Random, Forest’ Classifier.

a) Kd-Tree Classifier: Initially, we used the default KDTree parameters(leaf_size=40 and metric=’minkowski’), however, we did not get satisfactory accuracy with the default parameters. So we changed leaf_size to 20, it ran too long with this leaf size but we found some improvement in accuracy scores. KDTree has only two parameters which can be tuned as it is a simple baseline classifier, so after training our model for a few more leaf_size values, we set it to 15. After training the kd-tree, the tree was pickled in order to store its parameters which could later be used for testing the dataset.

b) Random Forest: We chose Random Forest to be our second classifier [3]. We used the RandomForestClassifier from the ensemble package of the scikit-learn. The parameter that we used to build our random forest is the number of estimators = 50, maximum depth = 20, minimum sample split = 3. We fixed the number of estimators to 50 because we did not find substantial improvement by increasing the number of estimator to 100. Another reason to keep it low was resource limitations, and the huge size of the training dataset. Intuitively, it makes sense to limit the depth of each tree in random forest to speed up the training and to classify. It also helps in better generalization, we also tried max depth = 10, but it reduced the accuracy of our model on both valid symbols as well as valid+junk symbols. We kept the minimum sample split low as we wanted to fill all pure samples at leaf nodes whenever possible.

To find the top predicted label(highest probability class) of the sample, we can use predict(X, y) function of the RandomForestClassifier, where X is the feature vectors and y is a list of corresponding labels. The top 10 predicted labels are computed by finding the probabilities of all the classes using the predict_proba() function of the RandomForestClassifier. We used the argsort function of the numpy library to return the indices of the probabilities in descending order given by predict_proba function. These indices are mapped are used to index into classes_ list of the RandomForestClassifier.

Experiments and Results:

We trained our dataset on the 70% split of the valid symbols and tested the Kd-Tree model on the 30% training dataset as well as on the 70% dataset. From this we observed that the accuracy that we obtained for the 70% training dataset is 100% and for 30% training dataset of valid symbols is 85.14%. We merged the valid symbols and the junk symbols and trained our KD - Tree classifier on 70% training dataset. This classifier was tested on 70% and 30% testing dataset of valid and junk symbols. We observed that the accuracy of 70% training dataset was 99.96% and of 30% was 78.6%. Similarly, the results that Random Forest gave for 70% training dataset was 98.85% (resubstitution) and for 30 % training dataset was 88.38% for valid symbols. For valid plus junk symbols the accuracy for 70% split was 96.63% (resubstitution) and for 30% split was 82.31%. For the 30% split (valid symbols) we got the best results from Random Forest which is 88.38%.

We see that the accuracy for 70% training dataset for valid symbols is 100% as we have trained our Kd-Tree classifier on 70% of the training dataset. This shows that the trained model is able to correctly classify each sample of the 70% training dataset correctly. The accuracy obtained for the 30% training dataset on only valid symbols is quiet fair. 21957 symbols in the dataset were correctly classified out of 25789 symbols. We see that there are few symbols which have a very high accuracy of recognition (greater than 80%) such as '(', ')', '+', '-', '2', '6', '=', etc. Few symbols have a recognition rate lower than 10%, such as the 'o', '}', '\prime', 'X'. We believe that few symbols have a high recognition rate because the kd-tree has been highly trained on these symbols and the ones with a lower recognition rate have less number of symbols and hence, the classifier was not able to recognize those symbols from the test dataset. When we observed the confusion matrix for top-1 symbols, we saw that the symbol which was highly correctly classified was '2' and 'CROSS'. We observe that the accuracy for valid plus junk symbols for Kd-Tree classifier is lower than the one with only valid symbols. This is because maybe the classifier has seen many symbols which were also considered as junk.

Random Forest gave a better accuracy than Kd-Tree for valid and valid plus junk symbols for the same feature set which was given to the Kd-Tree. Hence, random forest has worked better on the features which were generated for each point in the dataset. Even though Kd-tree has performed well on the features generated, Random Forest has outperformed Kd-tree. Further, we also see that Random forest has given a better accuracy than Kd-tree for TOP 2. In the TOP 1 confusion matrix, we see some common symbols in both the classification results of Kd-tree and Random Forest in which '3' is often misclassified as ')', 1 is classified as '(', etc. This shows that both the classifiers get confused between 3 and ')'; 1 and '('.

We can therefore say that the classifiers usually give good results when Junk is not added into the dataset. We can improve our classifier results by adding in more features and applying PCA (Principal Component Analysis) to select the best features generated.

References:
[1].L. Hu, R. Zanibbi, Hmm-based recognition of online handwritten mathematical symbols using segmental k-means Initialization and a modified pen-up/down feature, IEEE Xplore (2011).
[2] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html
[3]
http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifie
r.html