CSCI 727 - Pattern Recognition - Project Phase 01

Sanjay Khatwani (sxk6714@g.rit.edu)
Dharmendra Hingu(dph7305@g.rit.edu)

March 5, 2018

1 Design:

For this classification task we performed three type of preprocessing including Shifting the origin for the whole symbol, Normalization of coordinates and Interpolation between points. After this we extracted total of 42 features for every *.inkml* file. The description of all features in covered in following section. The baseline classifier is *kd-tree* and the non-baseline is *Gaussian Naive Bayes* [later abbreviated as GNB].

2 Preprocessing and Features:

Following are the preprocessing step that we performed,

1. **Origin-shift:** The origin of all the images was shifted to $(0, 0)$. This provides a uniformity to the character image such that all the characters start at 0 and it also makes the extraction of features simple and easy to imagine.

2. **Normalization:** The coordinates are normalized such that all the y-coordinates lie in the range $(0, 1)$ and all the x-coordinates are calculated accordingly based on the normalized y-coordinate value keeping the aspect-ratio same. This brings the x-coordinates in the range $(0, delta\_x/delta\_y)$. The $delta\_y$ is the difference between max value of y and min value of y. Similarly $delta\_x$ is difference between max value of x and min value of x.

3. **Interpolation:** Between every sequence of points in every trace, 6 new points are added using linear interpolation, which uses the slope and sampled x values to generate new y values. The choice of adding 6 points is, based on trial and error. The 6 points gave us the better results. Interpolating between points helps in getting a clear picture of the symbol and would help in feature extraction as well.

We divided the given training data into training and testing set with the ratio 70% and 30% for both valid symbols and junk samples preserving the class priors. For each of the model we use these data as one 70% valid symbols, 70% valid symbol plus junk symbol and whole set for training the model. Then we serialize it and de-serialize it to use on remaining 30% valid symbols, 30% valid symbols plus junk symbols and provided testing set. The way we get six serialized version of two models.

The information we get from the *.inkml* file is bunch of points that are drawn and the sequence in which they were drawn. This is not the good feature to feed to our models. So we need to derive some features from this on-line features. We have extracted 6 different type features from each symbol accounting to 42 element long feature vector. The extracted features are off-line features, meaning the features were derived from the on-line features. Here the on-line feature indicate the pixel values for each of the point and the timing information that represent sequence of traces. Following is a list of features that we extracted from each symbol:

1. **Directions**: $[1 - 3]$ This feature counts the number of direction changes in the symbol. We also use the first direction change and the last direction change in our feature vector.

2. **Aspect Ratio**: $[4]$ This feature adds the aspect information to the feature vector which is $(x\_max - x\_min)/(y\_max - y\_min)$.

3. **Number of traces**: $[5]$ This attribute adds the information about the number of traces in a symbol. This is important because it directly relates to how many strokes are there in the symbol.

4. **2-D Histogram**: $[6-30]$ This features accounts for the density of points in a $5x5$ grid. The original symbol is divided into 5 parts both vertically and horizontally. This gives us a grid of $5x5$. Number of points across all the traces is counted for every cell in the grid. This grid is flattened and a feature of length 25 is extracted that provides the rough density distribution of points on the symbol in the symbol space.

5. **Average Curvature**: $[31]$ This feature is useful in extracting the information about curves in the symbol. The implementation has been done as explained in [1]. The curvature information of every point is extracted as follows:

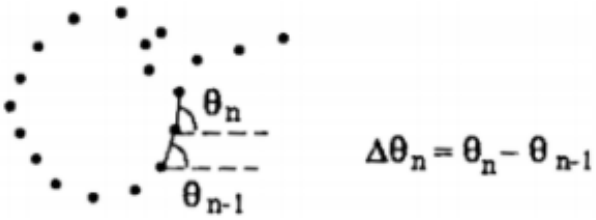
   (a) Calculate $delta\_x$:

$$delta\_x_t = \begin{cases} x_{t+2} - x_t & t = 0,\ 1 \\ x_{t+2} - x_{t-2} & 2 \le t \le N-2 \\ x_t - x_{t-2} & t = N-1,\ N. \end{cases}$$

   (b) Calculate $delta\_y$ similar to $delta\_x$.

   (c) Curvature =$arctan(delta\_y/delta\_x)$

   Curvature is calculated for every point and then an average over all values is returned as a feature.



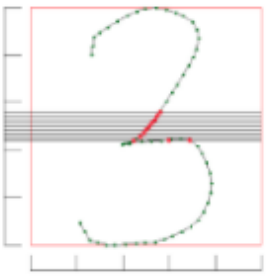
6. **Crossings**: $[32-42]$ This attribute provides information about the symbol as a whole. It is measured by counting the intersections between axially aligned lines assumed to be passing across the symbol. The implementation has been done as per suggested in [2]. The feature is obtained as follows:

   (a) x and y axes are divided into 5 equal regions each, giving in all 10 regions

   (b) For each region, 9 equidistant lines are considered. The number of intersections of the symbol for each of these assumed lines is counted. Average if these intersections gives the crossing feature of that region

   (c) Crossing feature for each region is calculated and returned. Thus, we get 8 crossing features that tell us about the average intersections in that every region

   A depiction of the crossing feature for a symbol is shown in the following figure. Each horizontal line in the center represents a sub-crossing and the red dots represent an intersection.



3 Classifier:

With the above extracted feature we use two different models to it them as training set to learn the model parameters and testing set to measure the accuracies for each of them. The baseline model we use is *kd-tree*. The implementation is provided by *scikit learn* library. The second model we use is *Naive Bayes Gaussian*. Again the implementation is provided by *scikit learn* library.

1. **kd-tree**: This uses nearest neighbor algorithm. This is unsupervised learning algorithm where we just pass in the feature vector and it learns the data on its own and returns us the cluster information. Now, when we query any arbitrary point in space the model returns k nearest neighbors to this point in space. The choice of k is up to the user but a word of caution is it may not return all the unique points. We would have to do some extra work to fetch all unique neighbors in order to get the top 10 list. The value of $k = 100$ worked for us. This provides the optimum trade of between performance and getting maximum number of unique classes in the overall results.

2. **Gaussian Naive Bayes (GNB)**: The main reason for choosing the model was the very fast training over all set of training features. Also, it is a probabilistic model and provides variety over other distance based models. This is different from the kd-tree in terms of making assumption about the distribution of data. The GNB assumes normal distribution of data. Since the extracted feature as obtained from the continuous feature space. On the other hand, kd-tree implementation uses distance based metric. In order to evaluate the GNB model, we use *model.predict_proba()* over testing set that returns the probability for each of the classes. The order in which the probabilities are returned is determined by another attribute *model.classes_*. Then we sort and route top 10 classes into output file.

The process of classification is rather straightforward. Simple snippet was generated to read in the .inkml file, then simply extracting the traces section using *xmltree* and plotting them helped visualizing the content. This visualization helped understand the given data and induced us to extract features described above. Next we divided the dataset into suggested 70% - 30% split. This was manually done for junk symbols as the given ground truth file and number of .inkml file were inconsistent. Then we extract the features from these .inkml file and route them to appropriate training or testing set based in the split. We save all the features as .npy file. This takes good amount of time. Now the process from here is easy we load the training dataset and fit the model (for both kd-tree and Gaussian Naive Bayes). After model is trained we save the model parameters to .sav file so that every time we dont have to train the model. For testing / validation we load the testing / validation set and saved model parameters and generate the output in .txt file. We produce top 10 predictions for the single sample in testing / validation set. For testing set we compute the classification accuracy and other evaluating measure using *evalSymbols.py* tool.

4 Result and Discussion:

For evaluating the classification rates returned by each of the model we store the prediction in a simple comma separated text file. The first column is the Unique Annotation ID of the .inkml file and rest follows the top 10 predictions produced for that model. We use provided *evalSymbols.py* tool to produce the confusion matrices, classification rate with or without junk and other insights.

Considering only valid symbol we get the following classification rate,

	Training Samples (Re-substitution)	Testing Samples (Evaluation)
kd-tree	99.47%	64.79%
Naive Bayes	53.92%	45.91%

Considering only valid symbol plus junk symbols we get the following classification rate,

	Training Samples (Re-substitution)	Testing Samples (Evaluation)
kd-tree	99.71%	62.88%
Naive Bayes	36.93%	37.27%

Few Observations: For kd-tree classifier:

1. The classification accuracy for this classifier is higher than the GNB classifier. This is because of the fact that it is an unsupervised techniques and learns the similarities and difference between features. On the other hand GNB uses class-conditional densities.

2. On the testing fold, we observe the mean position for correct classification at, approximately 2. This implies on an average the true classification of the samples occurred at position 2 for with and without junk.

3. The training accuracy for kd-tree is extremely high, this is expected since it should ideally be 100%, but this is compromised because of junk.

For GNB classifier:

1. The classification accuracy for this classifier is lower than the kd-tree classifier. This is because of the fact that Naive Bayes model assumes the data distribution is normal, which might not be the case for this data. Furthermore, we can say that classifier is not flexible in terms of capturing all the variations in different features.

2. On the testing fold, we observe the mean position for correct classification at, approximately 4. This implies on an average the true classification of the samples occurred at position 4 for with and without junk.

For samples, class *0* is misclassified with class
theta most of the time. This is due to people often write 0 with slash. Also, most of the time class *3* is misclassified with class *B*.

In the confusion matrix, (TOP 1, TOP 2 and TOP 3) we observe that many the diagonal is dense indicating we are doing good in terms of correct classification, but there is another density observed corresponding to junk classes both horizontally and vertically.

References:

1. H. Shu, "On-Line Handwriting Recognition Using Hidden Markov Models", Boston, 1997.
2. K. Davila, S. Ludi and R. Zanibbi, "Using Off-line Features and Synthetic Data for On-line Handwritten Math Symbol Recognition", ICFHR, Crete, Greece, 201, 2014.
3. Notes provided by Instructor on MyCourses: https://mycourses.rit.edu/d2l/le/content/686193/Home
4. http://scikit-learn.org/stable/documentation.html