BITS Pilani

CS F376

Design Project

Automation of Image Annotation Using Machine Learning Techniques for a Digital Archive of Paintings

Author:
Pratyush Kar

Supervisor:
Prof. Sundar B.

May 2, 2016

1 Introduction

With the development of internet and the reduction in cost of image capturing devices such as digital cameras, there is a rapid increment in the size of digital image collection. Efficient image searching browsing and retrieval tools are now becoming more and more indispensable. For this purpose many image retrieval techniques have been developed in recent times. All of these techniques are centred around two key domains:-

- Text Based Image Retrieval
- Content Based Image Retrieval

In text based image retrieval first the input image database is annotated with appropriate semantic tags. These tags can be identified using automated techniques but, in majority of the cases this has to be done using manual intervention. Once the annotated image database is obtained, image retrieval is done by querying the database system (DBMS) using appropriate querying techniques. One of the major issues with this approach is labelling the image database with appropriate semantic tags. To achieve a reasonable querying system, the quality of these semantic tags are of utmost importance. This is an extremely challenging problem and more often than not requires some form of human intervention. As the database size increases this technique simply doesn't scale. In CBIR, visual content of the images such as colour, texture, shapes are used to index the images. The fundamental difference between content-based and text-based image retrieval systems is that the latter requires manual intervention in some sense. Humans tend to associate high-level features in the form of semantic tags, in CBIR the algorithm tries to construct the high level features using low level features like colour, texture, shape, etc. Although there exist several sophisticated algorithms that try to capture high level features from low level features, these algorithms are unable to model the semantic representations in the human mind effectively [2]. Therefore, presently the performance CBIR is quite far from the user's expectations.

In [3] Eakins segregates the CBIR queries into three levels:-

- Level 1: Retrieval based on primitive features like color, texture, shape or the spatial location of image elements. A typical query in this scenario will be *'find pictures that look like this'*.
- Level 2: Retrieval of objects of a given type identified by derived features. For example *'find pictures of a flower'*.
- Level 3: Retrieval of images that contain a semantic relationship between identified objects. Query example *'find pictures of a man riding a horse'*.

Queries in Level 1 are typically a sample image to which the retrieval algorithm matches the images in the image database and returns the most relevant results. This form of querying is most restrictive as most search queries are typically text-based. Level 2 and 3 requires the development of high level features that powerful enough to bridge the *semantic gap*. Several of the techniques discussed in [1] try to achieve this by first separating the input images into blobs and then associate relevant semantic tags to these blobs.

This project is aimed at developing automated image annotation techniques for a digital archive of paintings. This is achieved by first segmenting the input image into local clusters based on colour and textural similarity. Then machine learning algorithms are used to classify the identified blobs into a number of high level semantic classes.

2 Graph Based Segmentation

Local clustering based on colour and textural similarity is done to separate the input image into blobs. This is achieved by applying a graph based image segmentation algorithm [4] on the input image. Graph-based image segmentation represent the problem in terms of graph $G = (V, E)$ where each node $v_i \in V$ represents each pixel in the image, and the edges in E connect pairs of neighbouring pixels. The edges are associated with an edge weight that is based on some property of the pixels it connects like colour intensity or textural similarity.

2.1 Algorithm

The input to the algorithm is a graph $G = (V, E)$, with n vertices and m edges. The output is a segmentation of V into components $S = (C_1, \ldots, C_r)$.

1. Create the image graph where V are the pixels in the image. For every pair of pixels in the 4 neighbourhood of a pixel add an edge e, where the edge weight is squared distance between the colour intensity of the pixels.
2. Sort E into $\pi = (e_1, \ldots, e_m)$ in the non-decreasing order by edge weights.
3. Initialise the segmentation S_0 where each vertex (pixel in the image) $v_i \in V$ is its own component.
4. Repeat step 5 for $q = 1, \ldots, m$.
5. Construct S_q from S_{q-1} as follows. Choose the q^{th} edge from π. Let v_i and v_j be the vertices connected by e_q.
   - If v_i and v_j belong to the same component. Do nothing.
   - If v_i and v_j belong to different components but $w(e_q) > threshold[v_i]$ or $w(e_q) > threshold[v_j]$, where $w(e_q)$ denotes the edge weight of edge e_q. The edge does not satisfy the merging criterion, do nothing.
   - If v_i and v_j belong to different components and the edge weight satisfies the merging criterion. Merge the components and update the threshold values.
6. Return $S = S_m$.

Figure 1: Graph Based Segmentation

3 Histogram of Oriented Gradients (HOG)

Histogram of Oriented Gradients (HOG) [5] is one of the most popular feature descriptor used for object detection in computer vision. The HOG descriptor technique counts the occurrences of gradient orientations in a given region of interest (ROI) and forms a histogram of gradient directions.

In HOG the image is divided into small connected regions called cells (generally of 8x8 size). For each of these of cells a histogram of gradient directions is computed. Within each cell gradient is calculated at each pixel location. Then these 64 gradients are taken and put into a 9 bin histogram. Histogram ranges from 0-180°(Dalal and Triggs used unsigned gradients), so there are 20 degrees per bin. For each gradient vector, it's contribution is dependent on its magnitude, so stronger gradient vectors have larger contribution. Furthermore, the contribution of gradient vectors is split into two nearest bins. For example, a gradient oriented at 75°, we add 3/4th its magnitude to the bin centred at 70°and 1/4th the magnitude in the bin centred at 90°. This is done so that slight changes in the gradient orientation don't change the histogram drastically (to ensure the stability in the histogram).

Broadly the HOG computation algorithm can be summarised (see Fig. 2) into the following steps:-

1. Compute centred gradients in the horizontal and vertical direction using 1D gradient mask [-1 0 1].
2. Divide the image into blocks (generally 16x16 size with 50% overlap).
3. Divide the blocks into further cells (generally 2x2 cells of size 8x8).
4. Quantize the gradient orientation, at each pixel in the cell, depending on gradient magnitude in the appropriate bins.
5. Apply contrast normalisation over overlapping blocks.
6. Compute the histogram across all cells and return the feature vector.

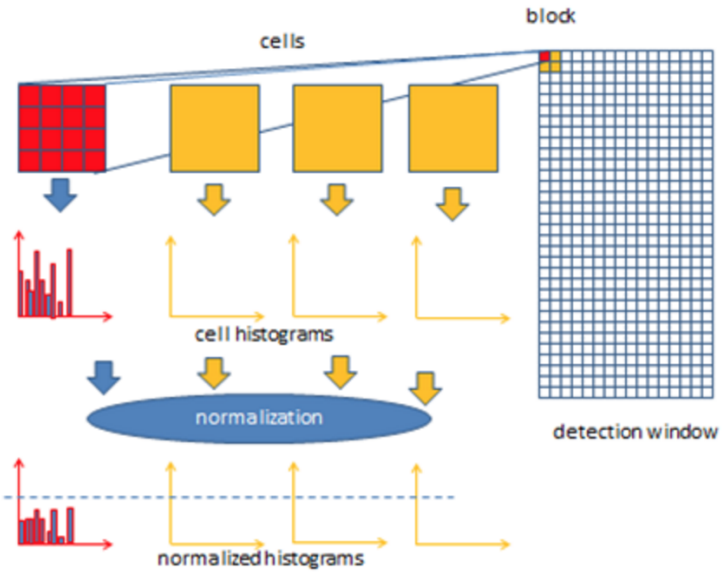

Figure 2: Histogram of Oriented Gradients

4 Support Vector Machine (SVM)

Support Vector Machine (SVM) [6] is a widely used technique for data classification. In this technique we try to find the hyperplane with the maximum distance from the positive and negative data points. This distance is termed as margin and the data points that form the decision boundary are called support vectors. The idea behind SVM is to find the optimal value of hyperplane parameters that maximise the margin of the decision boundary.

The primary reason why SVM is widely used is its ability to map the given data points to a higher dimensional space (maybe infinite) and apply SVM on the transformation without significant increase in the computation. This is done through the use of kernel functions K which allow dot product computation without actually transforming the original point to the higher dimensional space.

$$K(x_i, x_j) \equiv \phi(x_i)^T \phi(x_j) \tag{1}$$

Given a set of training data points (x_i, y_i), $i = 1 \dots l$ where $x_i \in R^n$ and $y_i \in -1, 1$, SVM tries to find the optimal solution to the following problem:-

$$\begin{aligned} \min_{\mathbf{w},b,\epsilon} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\epsilon_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\phi(x_i) + b) \geq 1 - \epsilon_i, \epsilon_i \geq 0 \end{aligned} \tag{2}$$

Since in a real world application the data is often not linearly separable, an error term is introduced in the optimisation condition. The constant $C > 0$ is termed as the penalty parameter and this needs to be set by the SVM algorithm.

The kernel function used in the implementation is the Radial Basis Function (RBF) given by the following equation:-

$$K(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}, \gamma > 0 \tag{3}$$

The values of these hyperparameters namely, C and γ is done by using the grid.py tool [8] provided in LibSVM [7]. This tool performs a "grid search" on C and γ using cross-validation.

5 Sliding Window Algorithm

Sliding window algorithm is one of the most commonly used algorithm for object detection in a digital image. The algorithm involves sliding a window of a predefined size across the image and applying the classifier at every instance. To detect objects of different sizes the input images is scaled appropriately and same algorithm is applied on the scaled image. The amount of displacement between two successive window instances needs to be initialised by the user and in our scenario the value needs to be a multiple of the cell size, in order to ensure compatibility with the HOG features. The second issue that needs to be addressed is a method to eliminate overlapping window detections. A simple way to do this will be to calculate the area overlap between overlapping windows, if the area overlap is above a certain threshold then the detections are merged.

5.1 Procuring the Dataset

The training dataset consists of 3 separate classes namely:

- Faces: 146 instances

- Flowers: 225 instances
- Others (Background): 1572 instances

Each data instance is a 64x64 RGB coloured image in the JPG format.

Figure 3: Sample images from the dataset

Each data instance was manually selected from the provided training images. If the data instance was not of the aforementioned size, the image was scaled to the appropriate size. The above figure shows some sample images from the data set.

5.2 Data Preprocessing

Each image in the training data set is converted to grayscale. To ensure symmetry in the data set, both the grayscale image and its reflection is added to the training image set. For each image in the training image set HOG features are computed and added to the training data set. The 64x64 pixel detection window is divided into 7 blocks across and 7 blocks vertically, for a total of 49 blocks. Each block contains 4 cells with a 9 bin histogram per cell, giving 36 values per block. Therefore the final feature vector size = 49 x 36 = 1764.

As the feature vector is very large the amount of training data generated will be massive which will result in slow training and detection time. Furthermore, high dimensional data suffers from the curse of dimensionality. As the number of dimensions increase, the feature vector space becomes more and more sparse. This affects the generalisation error of the classifier. To counter these issues we apply Principle Component Analysis (PCA) to extract new dimensions (principle components) that capture the maximum amount of variance in the dataset. It was seen experimentally that 30 principle components capture 82% of the total variance. Increasing the number of principle components to 75 did not increase the captured variance (87%) significantly. PCA allows us to reduce the number of dimensions in the training data thereby making the model less complex and reducing both the training and detection time.

5.3 Training the Classifier

The dataset generated in the previous set is partitioned into training and testing partition, in order to gauge the generalisation error of the trained model. The training partition is fed to LibSVM's SVM solver [7] and using the appropriate values of C and γ, the model is generated.

Using the above generated model the sliding window algorithm is applied to the input image. The amount of time taken to run the algorithm on an input image of size 2925x790 is around 5.5 seconds. The step that takes the maximum amount of time is the PCA transformation. We can optimise the time taken by parallelising the matrix multiplication step in the PCA transformation. The detections by the sliding window algorithm for a sample image is shown in the following figure in green.

Figure 4: Result of applying the sliding window algorithm.

6 Using the Code

All the dependencies have been installed on the provided machine. The following list is provided for reference:-

- OpenCV 2.4.9 or greater
- gcc, g++, cmake. Can be installed directly using *sudo apt-get install build-essential*
- Git for cloning the repository
- Python 2.7 with numpy and scikit-learn
- LibSVM ver. 3.21 or greater
- Eigen 2 (Linear Algebra library)
- gnuplot (optional) if the user wants to view the C versus $log(\gamma)$ curves)
- OpenMP (optional) to parallelise matrix multiplications in Eigen library

6.1 Segmentation

The graph based segmentation described in section 2 has been implemented using OpenCV in C++. To represent the various components in the segmentation, disjoint-set data structure is used. Union-by-rank is employed as the policy for the join operation and path compression has also been implemented to increase the efficiency of subsequent find operations. The components obtained after the segmentation step are post-processed to eliminate small components surrounded by large components. This is done by computing the size ratio between neighbouring components and if the size ratio is smaller than a user-defined threshold (COALESCE_RATIO), the components are merged into a single component. The directory structure of the segmentation module looks something like this:-

- convolve.h (Implements the convolution operations required for the gaussian blurring)

- disjoint-set.h (Implements the disjoint set data structure and its operations)
- filter.h (Implements the gaussian smoothing operation)
- image.h (Defines the image data structure)
- imconv.h (Defines image conversion methods)
- imutil.h (Various utilities)
- makefile
- misc.h
- pnmfile.h (File I/O for different formats of images)
- seg-graph.cpp (Driver script for the segmentation algorithm)
- segment-graph.h
- segment-image.h

The driver function provides several options.

- -c: Enable or disable post-processing of the components (default: OFF)
- -h: Display help message
- -k: specify the constant for the threshold function (default: 500)
- -m: Minimum component size, enforced by post processing stage (default: 15)
- -s: Constant required for the image smoothing (default: 0.5)
- -r: Provide COALESCE_RATIO (default: 0.02)

A sample call to the driver code will look like:

$$./seg\_graph < FILENAME > [-chkmsr]$$

6.2 Classification

The process of classification can be divided into 4 broad steps:-

1. Creating labelled dataset from the set of given training images.
2. Computing the HOG features and applying PCA in the data preprocessing step.
3. Training the SVM model. Finding optimal values of hyperparameters (C and γ) and using them to build the SVM model.
4. Using this model to run the sliding window algorithm in the input image.

Figure 5: Creating the labelled dataset.

6.2.1 Creating the labelled dataset

createLabelledDataset.cpp is a script that allows the user to browse through the images in the digital archive and selecting the training images. You can select the appropriate region of interest (ROI) using mouse input. The keys 'n' and 'p' can be used to switch between next and previous images and the scale of the region of interest can be adjusted using the '[' and ']' keys. '0' key can be used to reset the scale of the ROI. For convenience, the training sets for 'face' and 'flower' is already provided in the code repository.

6.2.2 Computing HOG features

createTrainingData.cpp computes the HOG features for the training images and their reflection across y-axis. It stores the feature vectors in a text file that will be used by LibSVM for training the SVM classifier. createTrainingDataPCA.cpp also applies PCA on the training data and stores the principle component vectors in a text file. Currently the number of principle components have been defined to be 30, but, this number can be changed by modifying the NUM_EIGEN_VECTORS variable.

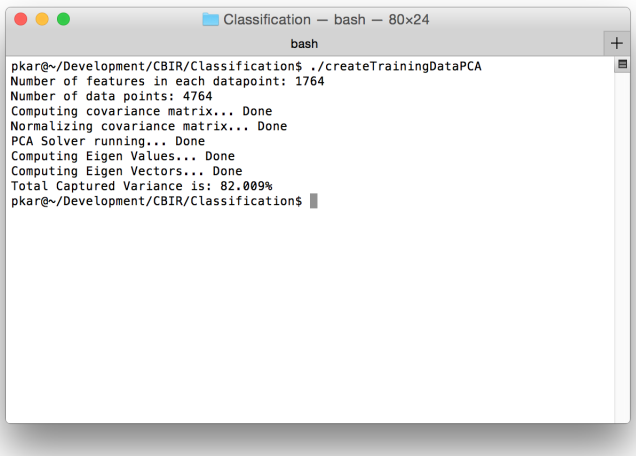

Figure 6: Computing HOG features.

6.2.3 Training the SVM model

generate_model.py is a python script to split the feature data into training and testing partitions and running the svm-train script of LibSVM to build the SVM model. The C and γ values calculated by the grid.py script should be edited in the train command for getting the best accuracy from the classifier. In our situation the optimal values of C is 8 and γ is 0.125. The training data set size can also be changed in the python script. The partitioning code ensures equal representation of both positive and negative samples so that the resultant model is not biased. To train additional classes, their class names can be added into the class_type list.

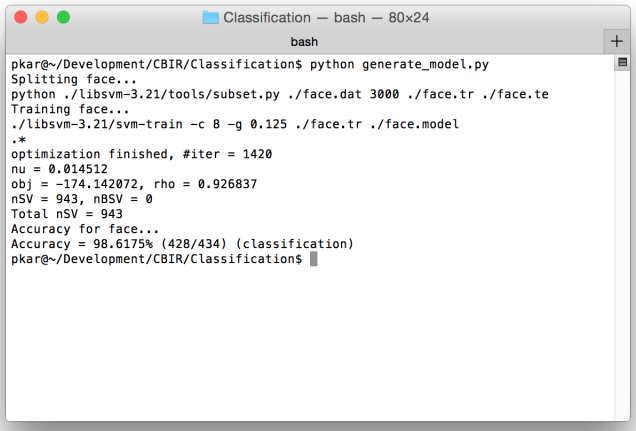

Figure 7: Training the SVM model.

6.2.4 Running the Sliding Window Algorithm

detectClassImage.cpp runs the sliding window algorithm on a sample image. The sample image path is given as a command line argument to the script. The sliding window stride length can be modified by adjusting the win_size parameter which is set to 8 pixels by default. detectClassImagePCA.cpp is similar in function but it additionally also computes the PCA transform for the window instances using the principle components computed in the HOG feature computation phase.

6.2.5 Additional Files

- HOG.h: Serves as a wrapper class to OpenCV's HOG implementation. Once the HOG features are computed for the entire image the user can directly access the required feature vector stored in the memory.
- PCA.h: Uses Eigen library to compute the PCA transformation for the feature matrix.
- svmPredictVector.h: LibSVM currently only provides functionality to predict new data points from files. This utilises existing LibSVM machinery to allow making predictions using feature vector stored in the std::vector<float> format.

7 Conclusion and Future Work

SVM allows us to create diverse classifiers based on the input features used. The complexity of the classifiers can be altered depending on the requirements. Although work still needs to be

done to come up with a method to rank the images based on the annotated tags. SVM gives us the class for a given data point, but it does not provide any measure of confidence. A method to rank the retrieved images based on their importance needs to be implemented.

One of the methods to achieve this is cross-media relevance models. Here we try to estimate the probability distribution between the annotated tags and segmented blobs. Although there is no one to one correspondence between the blobs and the tags, this technique provides heuristics to rank the images based on similarity. The work portrayed in this report builds the groundwork for such retrieval models and will help future works to develop such a retrieval system.

References

[1] Ying Liu, Dengsheng Zhang, Guojun Lu, Wei-Ying Ma. A survey of content-based image retrieval techniques with high level semantics.

[2] A. Mojsilovic, B. Rogowitz. Capturing image semantics with low-level descriptors, Proceedings of the ICIP, September 2001, pp. 1821.

[3] J. Eakins, M. Graham. Content-based image retrieval, Technical Report, University of Northumbria at Newcastle, 1999.

[4] Pedro F. Felzenszwalb, Daniel P. Huttenlocher. Efficient Graph-Based Image Segmentation.

[5] Dalal, Navneet, and Bill Triggs. Histograms of oriented gradients for human detection, in CVPR 2005. IEEE Computer Society Conference on, vol. 1, pp. 886-893. IEEE, 2005.

[6] C. Cortes and V. Vapnik. Support-vector network. Machine Learning, 20:273297, 1995.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011.

[8] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification. Technical report, Department of Computer Science, National Taiwan University. July, 2003.