CSC420 Project Report

Tal Friedman 999875899

December 7, 2015

1 Introduction

The problem that I'm going to be tackling in this report is that of identifying DVD covers in an image taken out in the world. Specifically I began with a library of 265 DVD covers, and a test set of 7 imzges, 6 with DVDs in the library somewhere in the image, and 1 without. In this report I will present how I was able to quickly and accurately identify which DVD was in each image, and localize where it was in each image.

2 Methods

There are a few steps to the pipeline that I used to identify DVD covers, which I will describe here.

2.1 Building a Vocabulary Tree using hierarchical k-means clustering

The main component to this project was building a vocabulary tree in order to be able to quickly find some reasonably good matches in our DVD cover library. The general premise of my implementation was to use the method outlined in Nister et al. [3]. In this approach, I began by using SIFT (in particular, the speedy implementation provided by OpenCV[1]) to get a series of feature descriptors for each image in my database. I then combined all of these descriptors, and ran k-means with $k = 10$ to get 10 subsets of the descriptors. I did this recursively to define a vocabulary tree - in this particular example we began with the root node, and each of these 10 splits defines a child of the root node. As in [3], I set a hard limit on the depth the tree was allowed to grow to, in particular I used $L = 5$. In total this gave my tree a maximum of about 110,000 nodes.

There are a few technical details here - firstly, I used minibatch k-means any time the amount of data to be split was $> 10,000$ descriptors which massively reduced the build time of the vocabulary tree. Additionally, if the number of descriptors to be split was smaller than the branching factor of the tree, then I stopped splitting even if the given node wasn't at the full depth yet. There was also the possibility that k-means could come up with a split of clusters such that one or more clusters had no associated training descriptors. In this case I simply removed these clusters, as with no associated training descriptors or images they would not be helpful at test time.

2.1.1 K-means

One of the few things I did not implement myself was k-means. Rather, I decided to use the implementation for k-means and minibatch k-means provided by scikit-learn [4]. I chose scikit-learn rather than doing my own implementation mainly for speed, as it is faster than my own pure python implementation would have been, and also has a built-in implementation of minibatch k-means, both of which have random restarting built into the code. Algorithm 1 shows pseudocode for generic k-means, and Algorithm 2 for minibatch k-means as described in [5].

2.2 Scoring

In order to make use of the vocabulary tree to query for the most similar DVD-covers to a given test image, I had to use some kind of scoring mechanism for a collection of feature descriptors. I decided to use the

Algorithm 1 pseudocode for k-means

```
means <- random points in space
while not converged:
        reassign each data point to the closest mean
        recompute each mean as the centroid of its assigned points
```

Algorithm 2 pseudocode for minibatch k-means

```
for i in 1..t:
        d <- subset of examples sampled from data
        assign each data point to the closest mean
        temp_cent <- means as centroids of assigned points in sample data
        cent <- weighted average of cent and temp_cent by # of points assigned to center
```

same approach as in [3] in defining the scores. Specifically, we define the score of an image s as an n dimensional vector $\vec{d}$, where n is the number of nodes in our vocabulary tree. Then, for each node i, we let $d_i = w_i n_{is}$, where n_{is} is the number of feature descriptors in s which pass through node i (Figure 1 gives a good illustration of this). As in [3], we let our weights $w_i = ln\frac{N}{N_i}$, where N is the total size of the library of DVD covers, and N_i is the number of DVD covers with at least one descriptor passing through node i. Thus, we see that inner nodes will tend to be downweighted, and leaf nodes make the largest contribution to the scoring vector. Finally, we normalize our vector $\vec{d}$ using an L1 norm, so as not to penalize or help images based on how many features they have.

To find the best matches for a query image in the database, we first compute a normalized score vector as described in the above paragraph, call it $\vec{q}$. We then simply consider the L1 norm of the difference between our vectors $||\vec{q} - \vec{d}||$ for each image in our database, and then take the images with the lowest such value. For my pipeline, I took the top 10 matches to continue on to the next step.

2.2.1 Precomputing database scores

Since our training set of DVD covers is reasonably small, the entire set was used to build the tree. This opens up the option to compute the score vectors for each image in the database of DVD covers as the vocabulary tree is being built. I did this by tracking the image of origin for each feature descriptor, so that as we construct a node i, we can also compute the score at node i for all images in the database. Algorithm 3 shows how nodes are built recursively, and how the scores are precomputed. Algorithm 4 shows how scores are computed for a new test image.

2.3 Homography estimation

The final part of this pipeline is, once the vocabulary tree has been used to retrieve the top 10 matches, to then do homography estimation on each of the 10 matches. To do this homography estimation, I began

Algorithm 3 Pseudocode for recursively building nodes, including precomputation

```
def build_node(data):
        N_i <- # of distinct images appearing in data
        w_i <- ln(N/N_i)
        node_scores[i,j] <- w_i * # of appearances of image j in data
        datas <- kmeans_and_split(data)
        for k in 1..10:
                build_node(datas[k])
```

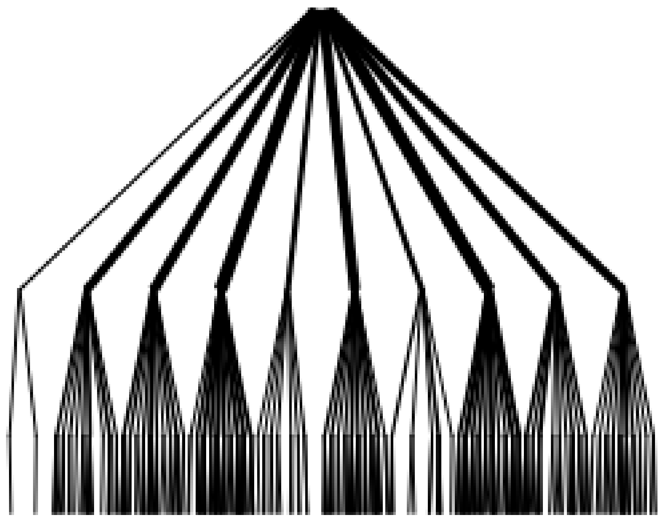

Figure 1: An illustration of the descriptors in a vocabulary tree with three levels for an image with 400 features. Taken from [3]

Algorithm 4 Psseudocode for computing the score of a new image

```
def score(i, data):
        scores[i] <- w_i * size(data)
        datas <- split(data)
        for k in 1..10:
                score(child_k, datas[k])
```

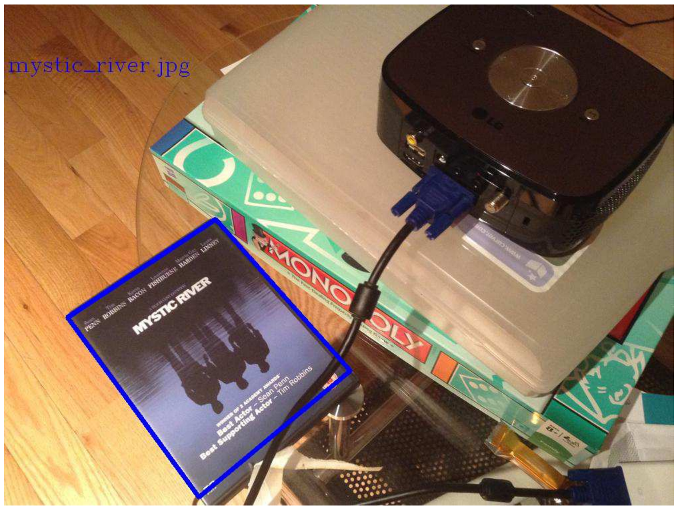

Figure 2: Correctly identified and located mystic river dvd case

by doing feature matching, getting the top match in the test image for each feature in the reference image, and then using Lowe's method for filtering, removing any matches where the ratio of the distances between top hit and second hit were > 0.7. Once I had these matches, I ran 1000 iterations of RANSAC, at each iteration picking 4 matches with distinct test and reference features, computing a homography matrix, and then counting inliers based off this matrix. The matrix that gave me the highest number of inliers was the one I kept, and I used the actual number of inliers to decide which of the top 10 matches for the test image I chose.

2.3.1 Transforming points with a homography matrix

The last thing which I did not implement myself was using the homography matrix to transform a point into the new space when doing RANSAC. This operation is a pretty simple matter of multiplying a matrix and a vector and then doing some normalization, but I decided to use the function built in to OpenCV for speed, since each iteration of RANSAC had to do this for many points, it had a large effect on the running time.

3 Results

3.1 Accuracy

Since our test set only consisted of 6 images with matches in the database, there weren't any meaningful statistics I could collect. My pipeline was able to identify the correct DVD from our library for each test image where it was possible, as well as come up with a generally accurate idea of where the DVD was in the image. Figures 2 and 3 a couple of the more difficult test cases, which my pipeline was able to identify and locate successfully. One interesting thing to note about these results is that, when built with $L = 4$, the vocabulary tree has a hard time finding the right match for one of the more difficult images. Figure 4 shows what happens in this case. Intuitively, this image is difficult as the DVD here is more zoomed out and difficult to see, as well as being on a heavier angle than many of the other test images. Even the colour scheme is different from the DVD image in our database, likely a result of the lighting in the scene where the picture was taken.

3.2 Running Time

I found that the vocabulary tree I was training with $k = 10$ and $L = 5$ took around 30 minutes to train. To deal with this, my code can also save and load models from disk, which takes between 5 and 10 seconds each time. In terms of actually running the pipeline on new test images, the initial query in the vocabulary

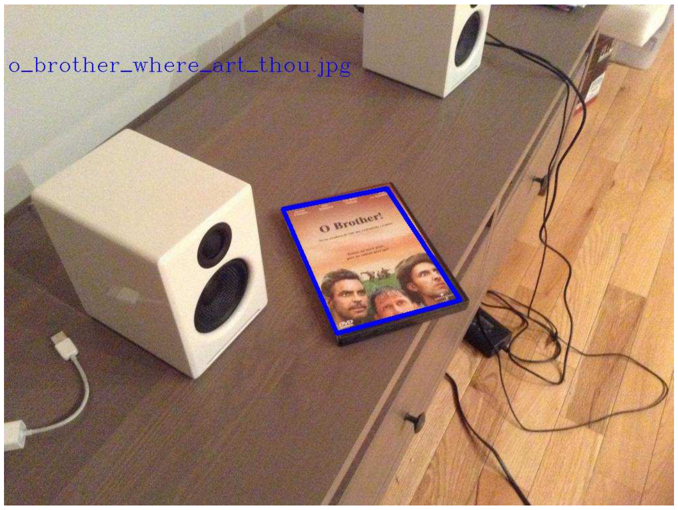

Figure 3: Correctly identified and located O Brother DVD cover

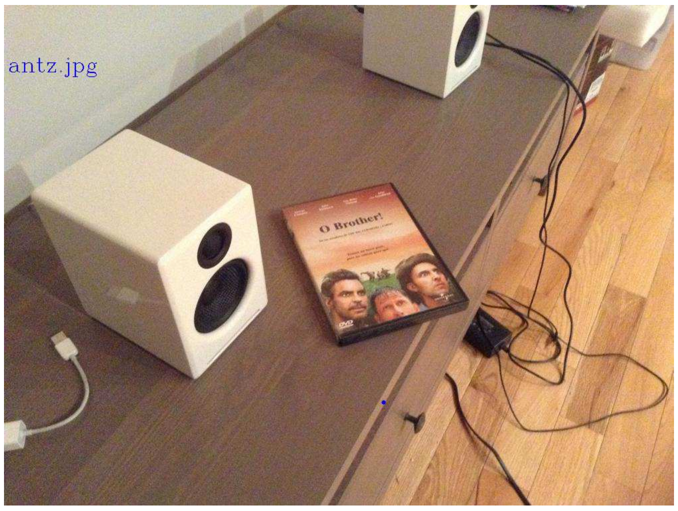

Figure 4: The vocabulary tree with $L = 4$ is unable to identify the correct DVD in a more difficult test image

tree for the top 10 items takes just under 3 seconds. Then, doing 1000 iterations of RANSAC on each of the 10 database images takes between 5 and 20 seconds for each image, where about 90% of that time is spent doing matching.

4 Potential Improvements

Although my pipeline worked quite well for the given training set of DVD covers and test set, there are a few things that could definitely be improved. One major improvement that could be made is in the feature matching step, since this is a large portion of the time consumption when doing homography estimation. Rather than using a naive approach where we try all combinations and then sort, we could do some kind of approximate nearest neighbours based matching, as described in [2].

Another potential issue comes with scalability. With a larger training set of DVD covers, we would not be able to use all of it to build the tree. This also means that we would not be able to precompute the entire matrix of training image scores, and so we would have to do be able to do this quickly after building the tree. To do this, we could use the reverse looks up and quick summation method for computing image scores as described in [3].

References

[1] G. Bradski. *Dr. Dobb's Journal of Software Tools.*

[2] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[3] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] D.. Sculley. Web-scale k-means clustering. *Proceedings of the 19th international conference on World wide web*, 2010.