Skyline Classifier - Part 2 Report

Benedict Becker

Methods

In this project, I tried a number of different methods with varying degrees of success. My three main strategies were:

- Deep Transfer Learning
- Bag-of-visual-words
- Brute Force Classifier (or BF2 - I'll explain)

1) **Deep Transfer Learning:**
   I had a transfer learning pipeline all set up and ready to go from my work on iris spoofing, so I thought it was worth a shot to try it on this problem. A big problem I went into was that my dataset had some images 10X bigger than other - a lot of different sizes and shapes. Unfortunately the original network, VGG, only took in images of size [224,224,3]. To get around this, I shrank down any extremely large pictures down to a reasonable size (preprocess.resize()), chopped off some of the top and bottom of each image (because the buildings were usually somewhere in the middle) and then cut my square out of the middle of that result.

   Since VGG accepted color images, I was able to keep all three channels. As with iris spoofing, I took off the top layer of VGG and added a 4-node softmax layer. Unfortunately the net wasn't able to learn anything significant from the images as my accuracies were barely above the 25% baseline accuracy. My guess is that shrinking those images so far down reduced the distinguishing features of each image. It was hard for me to tell the correct skyline by looking at the shrunken pictures, so I imagine the network had trouble as well.

2) **Bag-of-visual-words**
   This was the strategy I spent the most time on. Since there would be similar objects (the skyscrapers) in different parts of each image and at different sizes (different picture locations), not to mention different image sizes, keypoints seemed to be the way to go for this problem. To preprocess the images, I converted to grayscale, downsized extremely large images, and chopped off the bottom third of the image (this helped, as there were many keypoints found in the bottom third of the image and the buildings were rarely there). Then, I used ORB to find keypoints and descriptors. I used ORB instead of SIFT or SURF because my version of OpenCV didn't include implementations of either. However, since ORB did something similar, I thought that it would be a sufficient keypoint detector.
   After extracting descriptors from all the training images, I used KMeans to cluster all of these features (dim 32) into VOCAB_SIZE clusters, with each cluster representing a "word" in our visual vocabulary. KMeans started taking a long time, so I switched to MiniBatchKMeans for excellent improvements in speed and no decrease in accuracy.

After building the vocabulary, I went through the training data again and determined counts of each word in each training image. Then, I used the resulting histograms to train my classifier.

I used three separate classifiers:

a) SVM
   This was the most common classifier, so I decided to try it first. I experimented around with the kernel and found that 'poly' was the best option. For some reason other kernels, including the Chi2 kernel you recommended, kept classifying ~98% of the test data as the same class, despite tuning γ and C. Nonetheless, the polynomial SVM worked well.
b) Neural Network
   Since some visual words might be more important than others, I tried training a 2-layer MLP network (of size [64,16]) as a classifier. Unfortunately, this classifier did not perform as well as the others. Also, additional layers did not seem to make a significant difference.
c) K Nearest Neighbors
   I implemented a KNN classifier just in case a small value of n_neighbors helped classify images better, as pictures of the city from certain angles might lead to images to being very close in the feature space. This classifier performed almost but not quite as well as the polynomial SVM.

3) **BF2**

I was disappointed with my accuracy from the other strategies, so examined the results from bag-of-words and found that many of the ORB keypoints were noise, or weren't on the skyline but on something else in the image. This could deter my clustering by reducing the usefulness of my words. I decided to use a Brute Force classifier instead, as that would take very similar keypoints heavily into account rather than the noise. Hopefully, this method would find the keypoints on the skyscrapers, and match them up.
First, I made a dictionary of descriptors from each of the training images. Then, for each image in the test set, I computed that image's descriptors and then found all the keypoint matches between that image and each image in the training set using the cv2 Brute Force Matcher (Needless to say, this method doesn't scale well).
Then, for each training image, I summed the *N* closest keypoints to my test image using the metric 1/distance (because I wanted very close keypoints to count for more). Finally, I used the top *M* training images based on this sum to vote on which class it was.
Since I use a brute force search to find the best images, using a brute force keypoint matcher, I call this method BF2.
Thankfully, I got significantly better results with this method, so all my efforts weren't in vain.

Accuracies

Since my classes are unbalanced, I used the F1 score to assess my model, which takes both precision and recall into account. I used a weighted average on a per-class basis to get a single F1 score for each strategy.

Model	***F1 Score (Validation)***	***F1 Score (Training)***
Polynomial SVN	0.3940800874772509	0.996667996737
MLP Network	0.3627690593848029	1
K Nearest Neighbors	0.3847359310387199	1
BF2	0.533451424998	1

By a long shot, my best method is using the BF2 strategy. I computed more detailed metrics for the best method (a screenshot straight from the horse's mouth, as it were).

	precision	recall	f1-score	support
Shanghai	0.43	0.83	0.57	36
London	0.43	0.22	0.29	27
New York City	0.72	0.41	0.52	32
Chicago	0.67	0.60	0.63	55
avg / total	0.58	0.55	0.53	150

Some observations:

- These accuracies are lower than I thought I would get, so this problem is more difficult than I initially expected
- From my research, SURF and SIFT are better keypoint detectors than ORB is. I suspect using SURF or SIFT would improve my accuracy.
- London performed worst. I suspect that this is because the London skyline is very scattered and there are two separate 'skylines' that one might take pictures of (Canary Wharf and downtown, with the "Walkie Talkie" building and the Shard).
- Additionally, the other three cities have a large, distinct building that is probably prominent in each picture (The Shanghai Oriental Pearl Tower, the Willis (Sears) Tower and One World Trade)

- Since I used very powerful classifiers, it makes sense that they are near-perfect on the training data. My challenge is to generalize them to classify pictures they might not have as much experience with.

In the future, I have several routes of improvement:

- From what I can see, SURF and SIFT are better at finding keypoints, but they weren't in the version of openCV that I have (since they're both proprietary). There are implementations that I can build online, so I'll give that a shot and see if I can improve my accuracy.
- I could use an ensemble of the different classifiers I built for the bag-of-words.
- I could use an ensemble of the different keypoint detectors (ORB, SIFT, SURF)
- There are lots of hyperparameters and a relatively short training time, which means that a hyperparameter search could be beneficial.

Example of processed image: