Problem 1: Texture Analysis

Motivation

As we learned in class, the tricky part of telling one texture from another is determining a set of features that are indicative of texture distinctions. Intuitively, we tell apart textures by a variety of features including the reflective properties (brightness), and how bits of dots and lines are distributed across the patch.
Laws filters provide a way to quantify this intuition. In this assignment, we consider 5 first-order Laws filters:

Table 1: 1D Kernel for 5x5 Laws Filters

Name	Kernel
L5 (Level)	[1 4 6 4 1]
E5 (Edge)	[-1 -2 0 2 1]
S5 (Spot)	[-1 0 2 0 -1]
W5 (Wave)	[-1 2 0 -2 1]
R5 (Ripple)	[1 -4 6 -4 1]

As shown, each of the 5 filters capture a potential aspect of variation of the pixels. The outer products of these first-order filters with one another create 25 second-order Laws filters, which provides greater feature variety.
Given a texture image, its convolution with the 25 filters give 25 matrices. Stacked together by the third dimension, this means 25 features for each pixel. We wish to encode the texture image with the variances of these 25 features, so with the variance equation $Var(r) = E[(r - \bar{r})(r - \bar{r})^T]$, and knowing that all other 24 features except for the one convolved with $L5L5^T$ have zero means, we can get the encoding by first subtracting pixel-wise the first feature matrix corresponding to $L5L5^T$ by its mean, then doing pixel-wise square for each pixel and each feature, and finally taking the average for each feature across its matrix. This way we get a 25-D feature vector for each texture image.
Now that we have encoded each texture image with a set of indicative features, by which two instances of the same texture should be similar to each other and two of the different textures should be less similar, we can use various ML algorithms to classify textures.

Methods

Much about Laws filter has already been explained in Motivation, but I include two preprocessing steps to improve performance. The first is to use MATLAB's *adapthisteq* to contrast-enhance the texture image; this is expected to exaggerate the variations in the texture image and thus further distinguish one texture from another. The second is to standardize the texture image with its mean and standard deviation; this is to prevent some features, especially the first $L5L5^T$, blowing up and dominating the other features.
In order to quantify how much each feature contribute to classifying the texture image, intra-class variances of each feature are calculated with $\Sigma_i\Sigma_j(y_{ij} - \bar{y}_{i.})^2$ and inter-class variances as $\Sigma_i\Sigma_j(\bar{y}_{i.} - \bar{y}_{..})^2$, where i corresponds to observations and j to features. Then the discriminant power of a feature is calculated

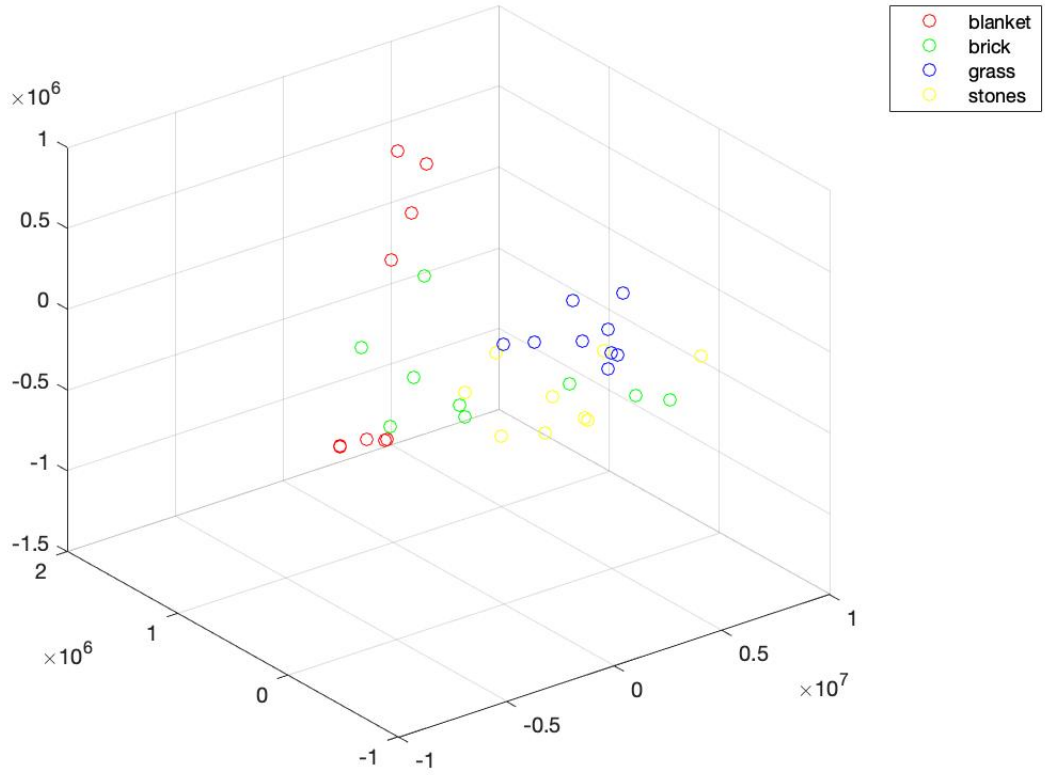
by dividing the intra-class variances by inter-class variances. The intuition here is that features that are discriminating should be similar within-class and dissimilar across classes; therefore, lower discriminant power value means more discriminating feature.
4 ML algorithms are attempted. The first is a very simple nearest neighbor classifier, i.e. the class of a test texture image is the class of its most similar texture image from training set. Since the 25 features have different variances, we want to correct the feature distances by their variances to avoid some feature distances dominating over others. For this reason we use mahalanobis distance instead of euclidean distance to determine the nearest neighbor. In addition, PCA is applied to reduce feature dimension from 25D to 3D, this helps avoid ML getting confused over the high number of feature dimensions while also helping visualization.
The second is kmeans classifier. Here I use MATLAB's *kmeans* function with correlation distance, i.e. feature distances are standardized when assigning instances to the nearest centroid. This distance measure is chosen for the same reason as using mahalanobis distance. kmeans is first used to cluster the 48 training images and obtain 4 centroids, and then the test images are categorized to their nearest centroids by standardized euclidean distance. Mahalanobis distance is not used here because of covariance matrix singularity, since there are 25 features but only 4 centroids.
The third is random forest. Here I use MATLAB's *TreeBagger* function with number of trees equal to 500.
The last algorithm is support vector machine. Here I use MATLAB's *templateSVM* with standardized features and polynomial kernal.

Results and Discussion

(a)

As shown, the second-order Laws filter that gives the lowest discriminant power value, i.e. the strongest discriminant power, is $L5L5^T$, followed by $L5E5^T$ and $E5E5^T$. $L5L5^T$ roughly corresponds to the reflective properties of the texture, i.e. how overall bright is the texture image. This aligns with intuition, since for example, blanket images tend to be brighter than brick images. $L5E5^T$ and $E5E5^T$ are also highly discriminating; this again aligns with intuition, since for example, brick images contain much fewer edges than the other texture images. The second-order Laws filter that gives the highest discriminant power value, i.e. the weakest discriminant power, is $W5S5^T$; this is probably because many of the images, regardless of texture class, contain granular pixels that exhibit this kind of variation. Shown below is PCA-reduced 3D feature plot:

L5L5^T	1.9885
L5E5^T	2.0417
L5S5^T	4.2704
L5W5^T	6.3004
L5W5^T	45.2663
E5L5^T	34.7584
E5E5^T	2.4396
E5S5^T	3.2578
E5W5^T	11.3634
E5R5^T	36.9337
S5L5^T	10.0601
S5E5^T	4.3808
S5S5^T	9.3710
S5W5^T	21.1163
S5W5^T	28.1210
W5L5^T	11.3155
W5E5^T	35.1941
W5S5^T	157.0536
W5W5^T	18.9935
W5R5^T	17.3139
R5L5^T	8.3407
R5E5^T	13.1991
R5S5^T	14.8629
R5W5^T	14.5816
R5R5^T	7.1526



The test accuracy rate of this procedure is 75%. Please run $prob_1$

(b) NOTE: The kmeans instructions here are somewhat confusing; I'm going with the implementation on piazza @387, i.e. apply kmeans to the training images, generate centroids, then use the centroids to classify test images. For this reason, the kmean purities shown below will be training purities.

(1) The results for kmeans can vary depending on how the centroids are initialized. The lowest error rate I've been able to get is 50%(purity $[0.3462, 1.0000, 0.8000, 1.0000]$) when using all 25 features and 58.33%(purity $[0.5000, 0.3889, 1.0000, 0.6667]$) when using only the top 3 PCA features. In general, PCA doesn't seem to affect kmeans accuracy much, which probably means many of the 25 features are actually redundant.

(2) Although random forest also has the element of randomness, I've been able to get 25% error rate consistently; my error rate for SVM is also 25%. In general, random forest seems to be more robust to feature scale differences than SVM; before I standardized the image, SVM used to be much worse than random forest. Random forest was also able to handle more features than SVM when I tried using more PCA features. However, random forest was more prone to overfitting.

Problem 2: Texture Segmentation

Motivation

This problem is similar to texture classification in problem 1, except now all textures are put into the same image, and the task becomes segmenting the image by textures.
Therefore the technique for calculating the image-wise feature variances by first mean-centering and then

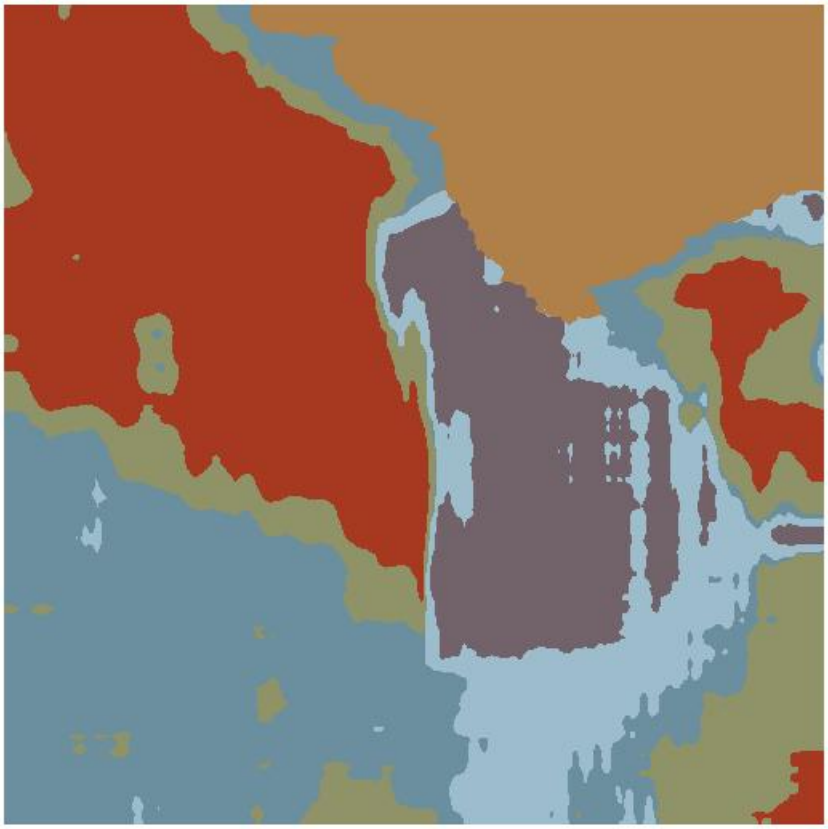
doing pixel-wise square no longer works. Instead, a mean filter is convolved with the image and each pixel's feature is the average of its neighbors. In this case, the size of the averaging window is pivotal in texture segmentation. Too large and smaller neighborhood patterns might be overwritten; too small and larger patterns might be missed.

Methods

The overall framework has already been outlined in the instructions. I made two passes through the image. The first pass generates the Laws filter responses. The second pass runs the mean filter. Like in problem 1, image was standardized before passing through the Laws filters. Further modifications will be saved for (b).

Results and Discussion

(a)



This result was generated with mean filter window size 61. Smaller window size would fail to capture the overall variation pattern and create many isolated blobs. Larger window size would merge different textures together.

(b) I made four modifications. First, I used *adapthisteq* to improve contrast of the image before passing through Laws filters; this helped distinguish the texture boundaries and exaggerate the features. Second, I used PCA to reduce the feature dimension from 24 to 3; this helped kmeans navigate the feature

space. Third, instead of mean-filtering the feature pixels, I tried using a gaussian filter to make each feature pixel sensitive to a neighbor's distance when averaging with it; here I used window size same as in (a) and $\sigma = 20$. Finally, I run a mode filter of window size 101 on the image to eliminate boundary stripes and isolated blobs. The resulting segmentation looks like this:

Problem 3: SIFT and Image Matching

Motivation

SIFT, or Scale Invariant Feature Transform, is a method that identifies and encodes keypoint blobs from an image in a way that's independent of scale and orientation.

In short, SIFT detects blobs by leveraging the observation that blobs are encircled by edges, and since Laplacian of Gaussian (LoG) filter creates a zero crossing point at each edge, with the brighter side of the edge getting a positive spike and the darker side getting a negative spike, LoG will create a peak at the center of the blob at a proper σ value. In addition, LoG with larger σ will peak within larger blobs, so σ also serves as a measure for how large the blob is.

Therefore, SIFT uses LoG of different σ to capture blobs of different scales. In practise, LoG is approximated with difference between derivatives of gaussian. If required, blobs can be normalized with σ, hence scale-invariant.

To account for the orientation of the blobs, SIFT divides each blob into sub-blocks, and calculates the

gradient of each block. A histogram can be created with bins being the discretization of 0 2π. The highest frequency gradient orientation can be considered the principle orientation of the blob, and the histogram itself, after being corrected by the scale and principal orientation, can be flattened to become a descriptor of the blob.
In addition to SIFT, this problem also explores BoW, or Bag of Words. After extracting the SIFT features from a training set of images, the features can be clustered with k-means to create a set of "words". At testing time, each feature can be assigned to its closest word and the test image be encoded with a "word" histogram. Then the histograms can be compared to determine which image from the training set is the test image most similar to, and hence classify the test image.

Methods

I'm using vl_sift from VLFeat. Before calling vl_sift, images are first converted from RGB to LUV; only luminance is passed to vl_sift. For all images, vl_sift is run with $peak\_thresh = 5$ and $edge\_thresh = 2$.

Results and Discussion

(a) (1) Scaling and rotation; paper mentions that although SIFT is fairly robust to affine transformations, it is not entirely invariant to it.

(2) To address scaling, SIFT rotates the feature descriptor by its principal orientation. For scaling, feature descriptors are generated with a constant number of sub-blocks, so the size of the blob doesn't matter. SIFT also uses (or rather approximates) scale-invariant LoG to avoid cases such as when large blobs give smaller peaks because of larger σ.

(3) Feature descriptor vector is normalized to unit length to address contrast change; constant change doesn't matter to feature descriptor since it consists of gradients.

(4) DoG is faster to compute; it also conveniently approximates the scale-invariant version of LoG as explained in (2).

(5) 128.

(b)

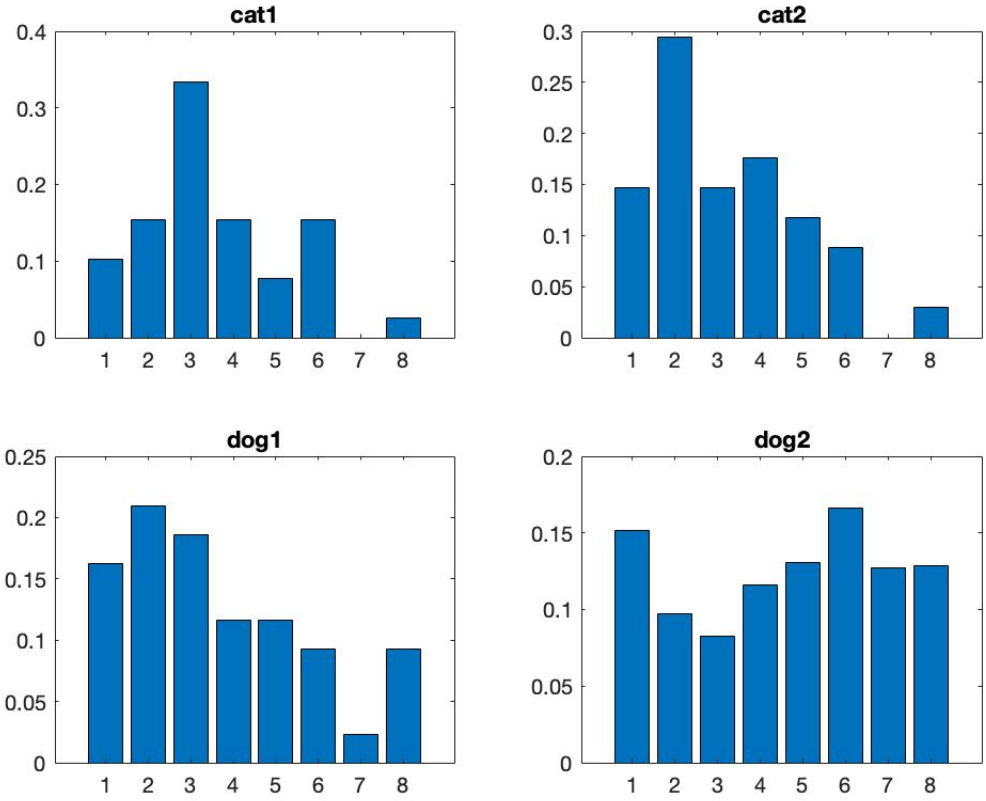
While the matching is not exactly intuitive, we should remember that SIFT only works with pixel values and doesn't do semantic matching. The SIFT blob with the largest scale on Cat_1 is the cat's forehead, which makes sense considering it's one of the few regions that is not blurred by focus and therefore has more salient edges. The forehead area has a few vertical stripes, which is probably why the orientation of the blob is horizontal. Counter-intuitively, the closest match in Cat_Dog is the dog's nose instead of the cat's forehead. But if we zoom in, we can see that the stripe patterns on the cat's forehead in Cat_Dog are quite different from those in Cat_1. On the other hand, if we zoom into the dog's nose, we see that there are lines perpendicular to the blob's orientation that correspond to the boundary between the dog's black nose and white fur. While it's still not similar, I can see why they can be matched.

These two blobs are matched probably because of the similar gradient changes between the crease in Dog_1 and the face in Dog_Cat. In the crease the pixels get darker going from bottom-right to top-left; in the face they get darker from top-left to bottom-right. Rotate by the blob orientation and they align.

This is an odd match considering the two cats have similar stripes on the forehead, so ideally SIFT should match to the forehead in Cat_2. However, notice that the head in Cat_2 is not as salient as in Cat_1, since the colors get mixed up with the cat's belly and body. I would actually argue that without external knowledge, it's difficult to tell apart the cat's head from the rest of its body in Cat_2 - and indeed, in this case SIFT doesn't even recognize a blob in Cat_2 forehead.

I think there's no need for explanation here - there isn't anything to be matched. Dog_1 image doesn't have the stripes in Cat_1. The match shown above is chosen among a set of bad matches.

(c)



Again, kmeans give different results depending on centroid initialization. This is one of the "better" runs and the similarities I get between Cat_1-Dog_2, and Dog_1-Dog_2 are 0.199 and 0.7219 respectively. [1] I run the script a few times, and although sometimes Cat_1-Dog_2 gets closer similarity than Dog_1-Dog_2, most of the times Dog_1-Dog_2 is considered more similar. I believe the matching is not as robust as it could be because Dog_2 includes a large background that is also rich in features, and this particular run performed well precisely because it didn't cluster the blobs that tend to correspond to background (histogram bin7). This can be confirmed by checking the SIFT blobs in Dog_2 - as shown, many blobs in the background.

[1] Here I go with piazza @444 and use only Cat_1, Cat_2, Dog_1, and Cat_Dog to generate the centroids. Dog_2 features are generated separately using standardized euclidean distance nearest neighbor.