Forming and using patch dictionaries

Idea:

- other patches in an image are good for denoising
- why not use other patches in other images?

Issues:

- how to find the right patch in a very large number of patches?
- redundancy
  - many patches are like one another, and so just make work

Desirable outcome

Procedure to

  accept a patch

  produce a patch/some patches that are similar

Built out of very large number of patches

Clustering

The key engine here is *clustering*. Clustering is a procedure that takes individual data items – in this case, patches, but others will appear – and produces blobs or *clusters* consisting of many similar data items. Each cluster has some representation in terms of parameters. You must determine (a) how many clusters there are; (b) what the cluster parameters are; and (c) which data items belong to which cluster.

Clustering

In **agglomerative clustering**, you start with each data item being a cluster, and then merge clusters recursively to yield a good clustering. Agglomerative clustering needs a good inter-cluster distance to fuse nearby clusters. There are three recipes for inter cluster distances. The distance between the closest elements tends to yield extended clusters and is known to statisticians as *single-link clustering*). The maximum distance between an element of the first cluster and one of the second tends to yield rounded clusters and is known to statisticians as *complete-link clustering*. The average of distances between elements in the cluster also tends to yield rounded clusters and is known to statisticians as *group average clustering*.

In **divisive clustering**, you start with the entire data set being a cluster, and then split clusters recursively to yield a good clustering Divisive clustering needs some criterion for splitting clusters. Mostly, this recipe is not much used in vision applications (though you will see echoes of the idea below) and I will not pursue splitting criteria.

Dendrograms

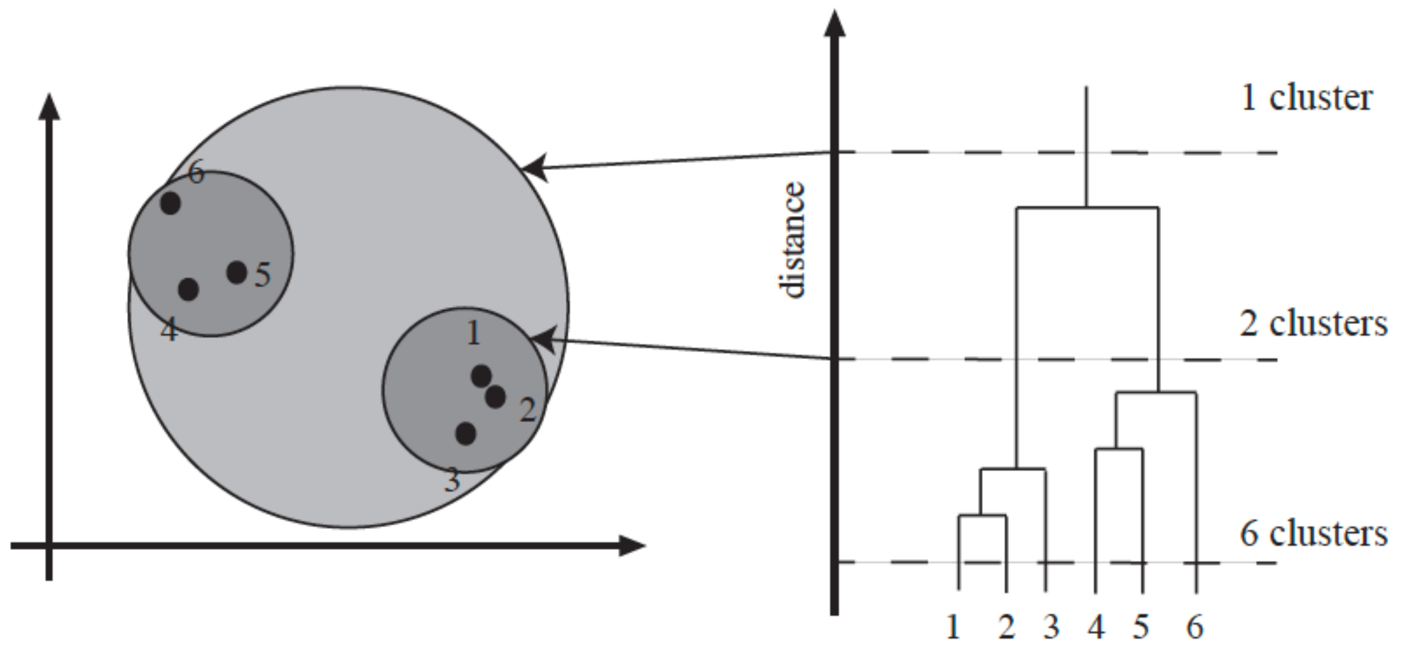


FIGURE 10.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

K-means clustering

Write $\mathbf{x}_i$ for a set of N image patches, straightened into vectors for convenience. Assume you know that there are k clusters. Write $\mathbf{c}_j$ for the center of the jth cluster. Write $\delta_{i,j}$ for a discrete variable that records which cluster a data item belongs to, so

$$\delta_{i,j} = \begin{cases} 1 & \text{if } \mathbf{x}_i \text{ belongs to cluster } j \\ 0 & \text{otherwise} \end{cases}$$

Every data item belongs to exactly one cluster, so $\sum_j \delta_{i,j} = 1$. Every cluster must contain at least one point, so $\sum_i \delta_{i,j} > 0$ for every j. The sum of squared distances from data points to cluster centers is then

$$\Phi(\delta, \mathbf{c}) = \sum_{i,j} \delta_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right].$$

Notice how the $\delta_{i,j}$ are acting as "switches". For the i'th data point, there is only one non-zero $\delta_{i,j}$ which selects the distance from that data point to the appropriate cluster center.

Minimize the objective function, but how?

You could cluster the data by choosing the δ and $\mathbf{c}$ that minimizes $\Phi(\delta, \mathbf{c})$. This would yield the set of k clusters and their cluster centers such that the sum of distances from points to their cluster centers is minimized. There is no known algorithm that can minimize Φ exactly in reasonable time. The $\delta_{i,j}$ are the problem: it turns out to be hard to choose the best allocation of points to clusters.

Approximate minimization

There is a remarkably effective approximate solution. Notice that if the $\mathbf{c}$'s are known, getting the δ's is easy – for the i'th data point, set the $\delta_{i,j}$ corresponding to the closest $\mathbf{c}_j$ to one and the others to zero. Similarly, if the $\delta_{i,j}$ are known, it is easy to compute the best center for each cluster – just average the points in the cluster. So iterate:

- Assume the cluster centers are known and allocate each point to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster.

Choose a start point by randomly choosing cluster centers (there are better options, below), and then iterate these stages alternately. This process eventually converges (**exercises**). It is not guaranteed to converge to the global minimum of the objective function, however. In this form, it is also not guaranteed to produce k clusters, but this is easily fixed, below. This algorithm is usually referred to as *k-means*

Fixes – efficient starting

One natural strategy for initializing k-means is to choose k data items at random, then use each as an initial cluster center. This approach is widely used, but has some difficulties. The quality of the clustering can depend quite a lot on initialization, and an unlucky choice of initial points might result in a poor clustering. One (again quite widely adopted) strategy for managing this is to initialize several times, and choose the clustering that performs best in your application. Another strategy, which has quite good theoretical properties and a good reputation, is known as *k-means++*. You choose a point $\mathbf{x}$ uniformly and at random from the dataset to be the first cluster center. Then you compute the squared distance between that point and each other point; write $d_i^2(\mathbf{x})$ for the distance from the i'th point to the first center. You now choose the other $k-1$ cluster centers as IID draws from the probability distribution

$$\frac{d_i^2(\mathbf{x})}{\sum_u d_u^2(\mathbf{x})}.$$

Choosing K

Strategies:

Cost function for various values of K [DOESN’T WORK!!!]

Purity, etc. [Hardly ever relevant]

Choose some K’s, try application with different K’s,
choose the best

Fixes – Scattered Points and Junk Clusters

If you experiment with k-means, you will notice one irritating habit of the algorithm. It almost always produces either some rather spread out clusters, or some single element clusters. Most clusters are usually rather tight and blobby clusters, but there is usually one or more bad cluster. This is fairly easily explained. Because every data point must belong to some cluster, data points that are far from all others (a) belong to some cluster and (b) very likely "drag" the cluster center into a poor location.

There are ways to deal with this. If k is very big, the problem is often not significant, because then you simply have many single element clusters that you can ignore. It isn't always a good idea to have too large a k, because then some larger clusters might break up. An alternative is to have a junk cluster. Any point that is too far from the closest true cluster center is assigned to the junk cluster, and the center of the junk cluster is not estimated. Notice that points should not be assigned to the junk cluster permanently; they should be able to move in and out of the junk cluster as the cluster centers move.

Fixes – Empty Clusters

- Assume the cluster centers are known and allocate each point to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster.

So some clusters might be empty and mean fails

Cure:

for any empty cluster, randomly select a data point to serve as its center

Why K-means is useful

Take input patch

Find its cluster center

Use patches in the cluster or the cluster center itself to denoise the image

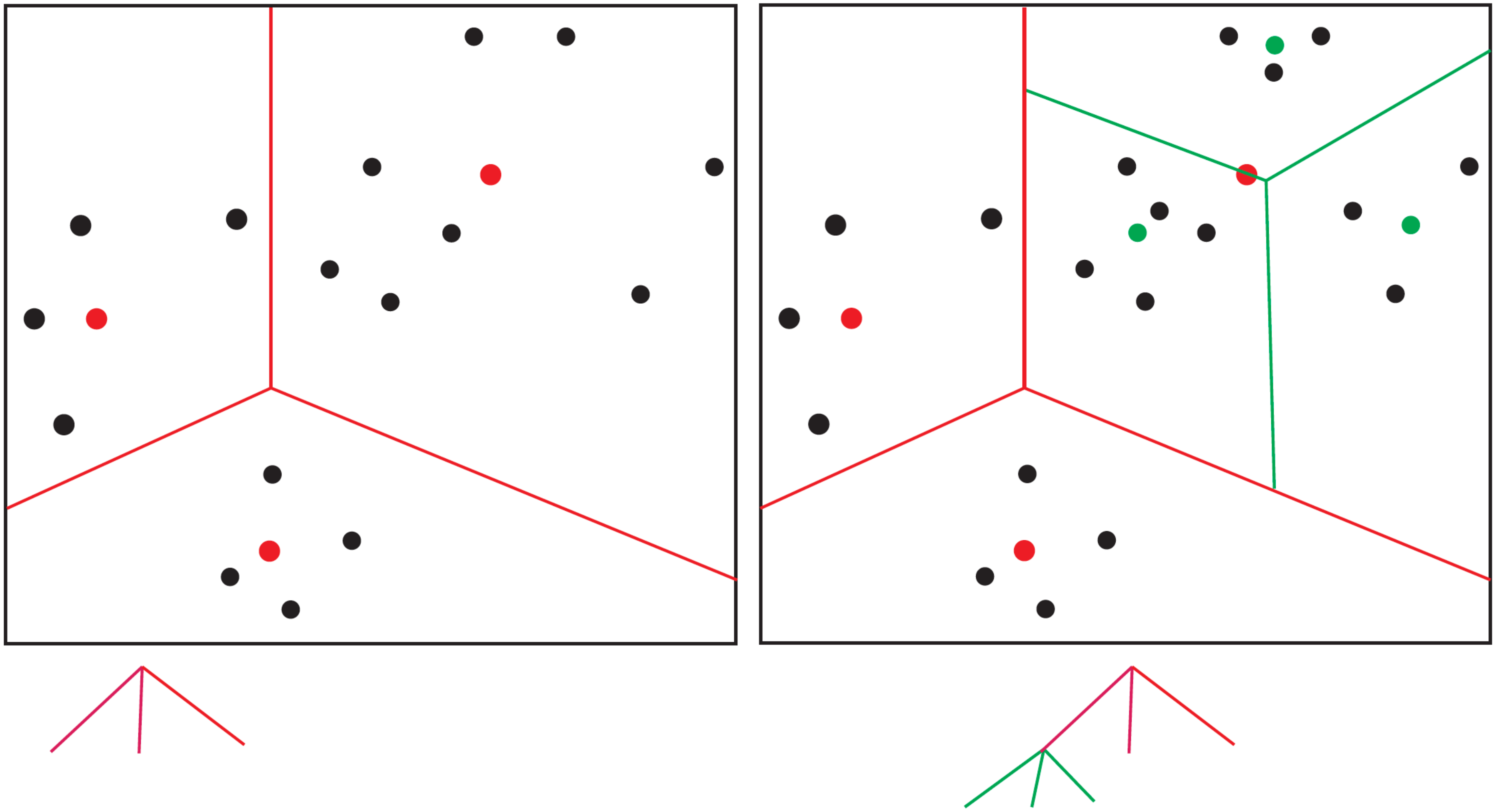
Hierarchical K means

But what if there are too many data points?

- too many clusters, finding cluster center is hard

IDEA:

- Cluster data using K means
- Each cluster is now a dataset
    - Cluster each dataset using K means



Hierarchical K means

But what if there are too many data points?

- too many clusters, finding cluster center is hard

IDEA:

- Cluster data using K means
- Each cluster is now a dataset
  - Cluster each dataset using K means

Notice this is a simple tree; you could make it deeper.

Nearest neighbors

Idea:

Find the patch that is closest to query patch, denoise (etc) with that

Issue:

How to do this fast? (v. hard)

Approximate nearest neighbors and k-means

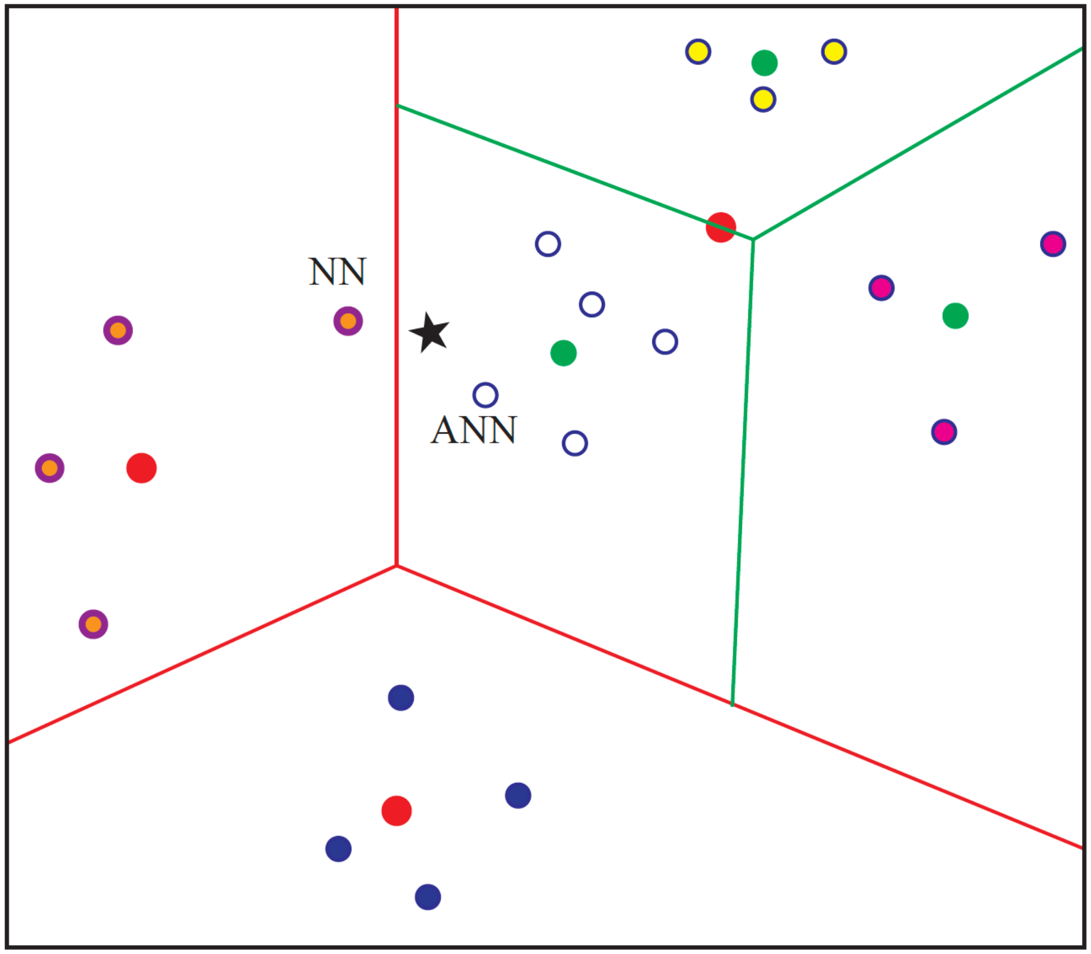
Idea:

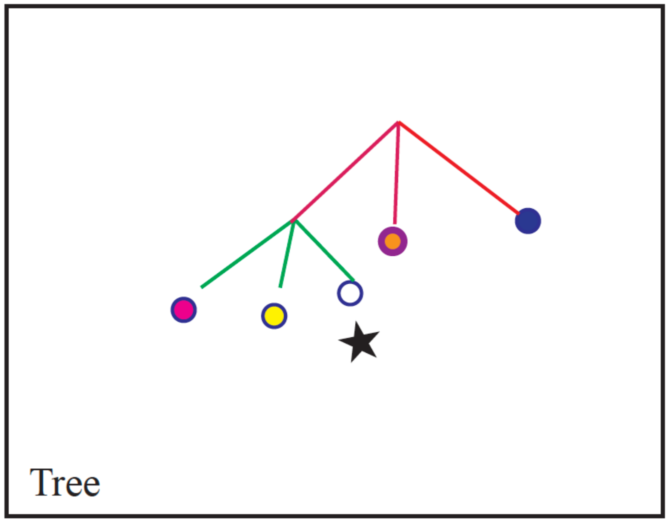
“Walk” input patch down hierarchical k-means tree

Find closest patch from patches in the leaf cluster

Use that instead of nearest neighbor

Approximate nearest neighbors and k-means





Query ★

Approximate nearest neighbors and k-means

Idea:

“Walk” input patch down hierarchical k-means tree

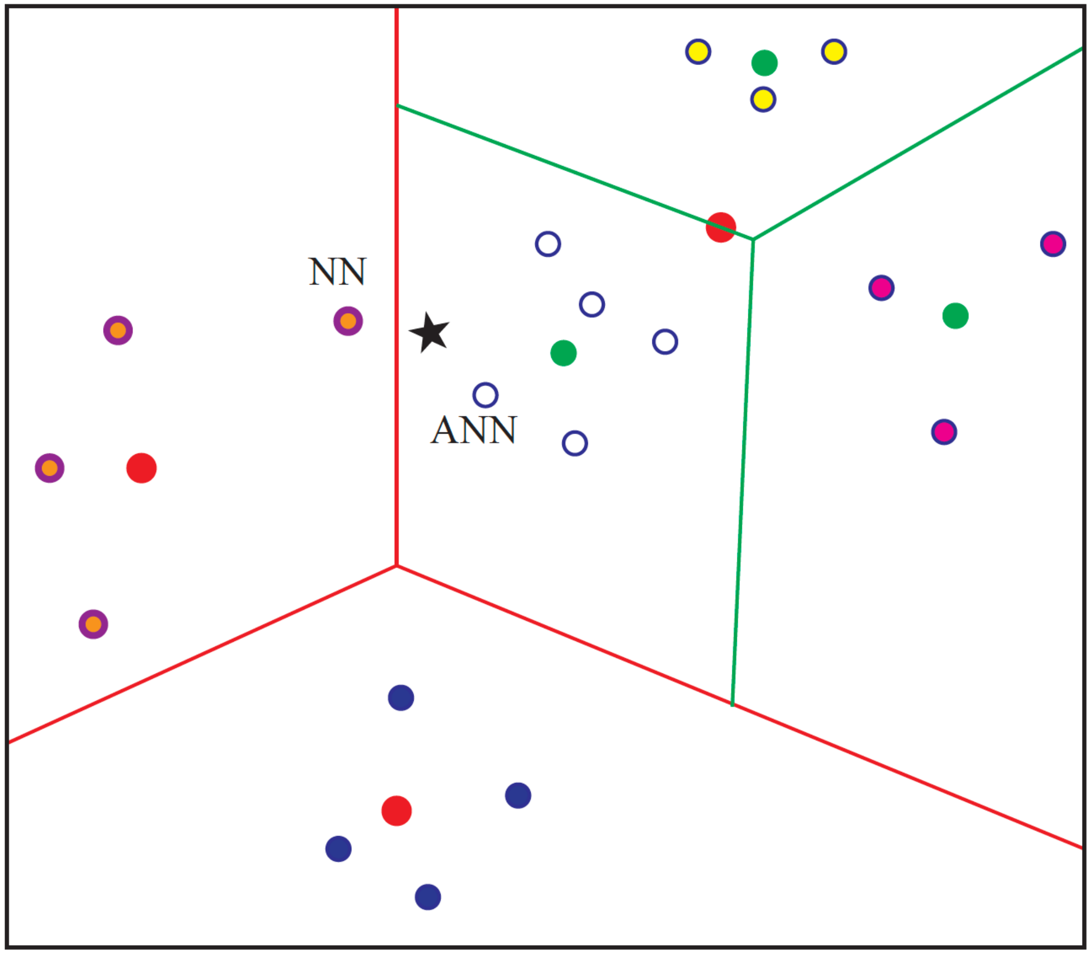
Find closest patch from patches in the leaf cluster

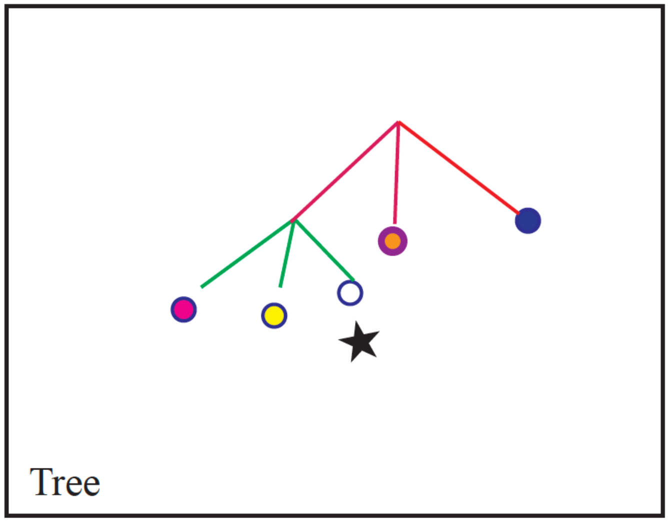
Use that instead of nearest neighbor

Fact: not nearest neighbor; with high probability nearly as close

Fact: hard to find a closer neighbor by walking tree

Approximate nearest neighbors and k-means





Query ★

Denoising with a patch dictionary

Procedure:

- Divide noisy image into patches, which could overlap
- Match each patch using a dictionary (ANN using HKM)
- Reconstruct
    - if the patches don’t overlap, easy
    - if they do, average

Options when matching

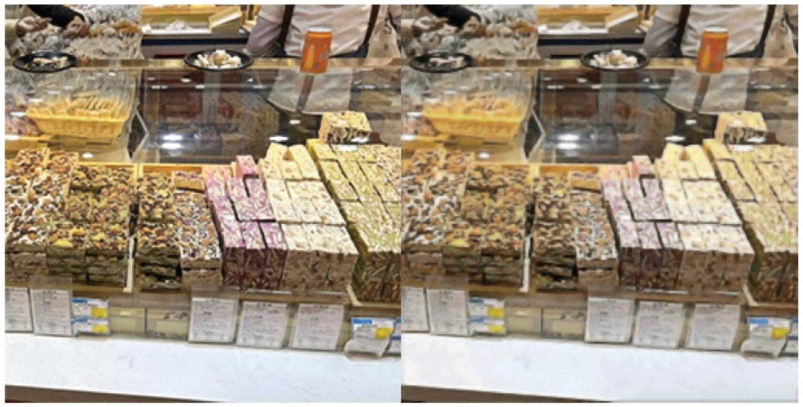
Walk tree, find patch in leaf that is closest to query patch

Walk tree, use mean of all patches in leaf

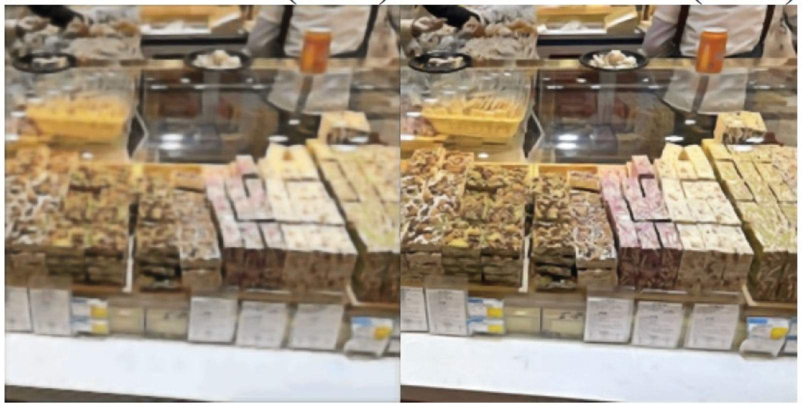
- this should be better than you might think

  cause all the patches should be quite similar

Denoising with a patch dictionary - overlapping



3x3, dictionary 1



5x5, dictionary 1

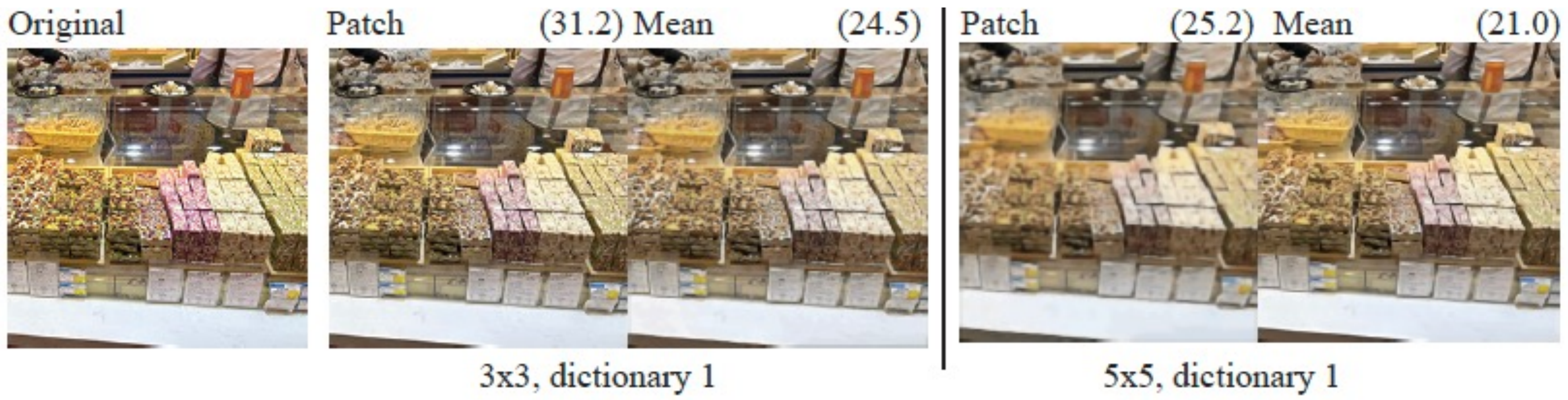
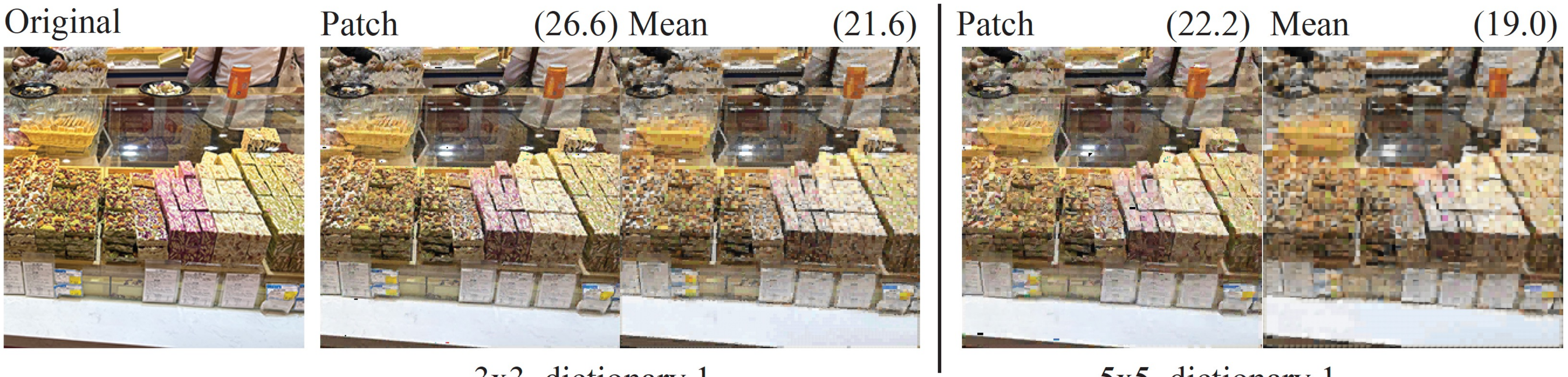


FIGURE 10.4: *Patch based reconstructions using patches from large image dictionaries can be very successful. On the* **left**, *the original image. Others show reconstructions using the approximate nearest neighbor (***Patch***) or the mean of the query cell (***Mean***) using patches of the size indicated. For these reconstructions, the patches overlap and are averaged (the stride is always* 1*). The PSNR of the reconstruction is shown in parentheses for reference. In each case, the tree contained 1e7 patches, taken from approximately 200, 000 images in the ImageNet test dataset (Section* ??*), and contains no patches from this image.* Image credit: *Figure shows my photograph of a sweetshop in Beijing.*

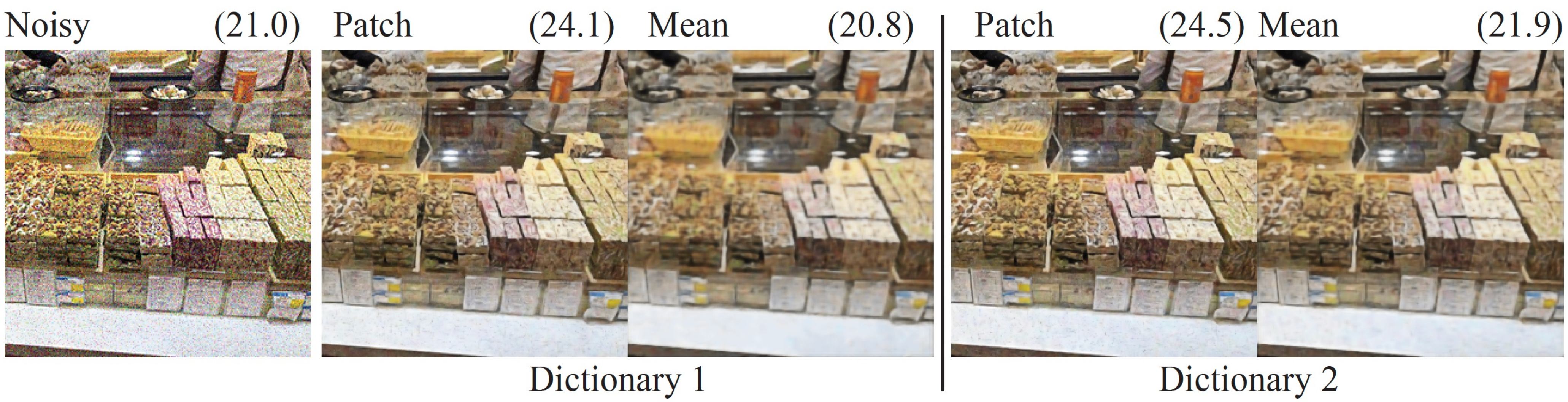
Denoising with a patch dictionary, no overlap



3x3, dictionary 1

5x5, dictionary 1

Dictionary size has a significant effect



Dictionary 1 – 1e7 patches, 200, 000 images, 2000 leaves

Dictionary 2 – 5 e 7 patches, 1e6 images

Vector quantization

Recall:

Walk tree, use mean of all patches in leaf

- this should be better than you might think

cause all the patches should be quite similar

Notice that you have built a mapping. You could replace the mean with the number of the leaf to get: patch-> leaf

This has a range of uses – compression, etc.

Building features with vector quantization

Here is a somewhat older construction that remains useful because ti can be so widely applied. Choose a patch size and some appropriate number of cluster centers. Represent the signal by cutting it into patches of fixed size, and turn these into vectors. Cluster a large set of training vectors. Now use the resulting set of cluster centers to represent a test signal by first vector quantizing each patch in the test signal, then building a histogram of the cluster centers. This histogram has fixed size (the number of cluster centers), and so can be used in a linear classifier.

Image might have variable size

Histogram has fixed size

Idea: vector quantization and voting

Problem:

you want to classify an image

Strategy:

learning:

- get many labelled images
- cut into patches
- build tree
- for each leaf, record the most common label

classifying:

- cut image into patches
- pass patch down tree
- vote for label

Now old-fashioned, but moderately effective and very good in its time