Anthony Poerio (adp59@pitt.edu)

CS1674: Homework 5 - Written

Due: 10/3/2016, 11:59pm

QUESTIONS

1) **When matching features across two images, why does it make sense to use the ratio: [(distance to best match) / (distance to second best match)] as a way to judge if we have found a good match?**

   Firstly—from a Euclidean-distance perspective—it makes intuitive sense that if ***both*** of the *two best* matches in different features we are comparing have a very similar distance ratio, then the features we are comparing have a similarity.

   Second, because we are taking a ***ratio***, and not simply measuring the distance itself, we are comparing an attribute that is **invariant to scale and orientation changes**. This means we have a higher likelihood of finding a match, even if we are looking at similar features from a different perspective.

   Third, by comparing the ratios of the best two matches, we can perform a matching computation in a **highly efficient manner**. So we can perform more comparisons overall, in less time, further increasing the likelihood of finding a match simply because we can 'search' across more data points.

2) **How do we use clustering to compute a bag-of-words image representation? Describe the process.**

   First we separate the image into patches/regions of interest. Next, compute a 'descriptor' for each region, based on its attributes. Each of these descriptors can be thought of as a point in some high-dimensional 'feature-space'.

   From there, we know that if points that are 'close' to each other in this higher-dimensional space must have similar descriptors, and therefore similar content. Similar to how a dictionary works for text, we can then 'index' into features with similar content by performing the same feature-parsing computation on the point we are interested in, and seeing what other features are close-by, in the higher dimensional space we have defined. Because this process is inspired by looking up similar words (similar based on morphology) in a dictionary, we call this a 'bag of words' representation of the image.

The ‘**clustering**’ aspect of this occurs when we *quantize* the feature space according to some algorithm (considered a black box at this point in the course), and ‘tag’ all the images in the quantize space we have defined as holding the attributes of some generic feature. Using this method, every time we compute features in a new image, we can say with some degree of certainty that this feature is ‘like’ other features we have seen before. We call these features ‘words’, and assign them by determining the closest cluster center after performing our feature parsing computation.

At the end of this process, images themselves can be represented in the ‘bag of words’ style by *summarizing the distribution* of ‘visual words’ using a **histogram**.

3) **How can we find to which cluster we should assign a new feature, which was not part of the set of features used to compute the clustering?**

After the initial words themselves have been computed and quantized, we can then assign ***new*** features to the ‘word’, or ‘cluster’ that is ***closest*** to our new feature in the higher-dimensional space we have designed. Essentially, this is still an exercise in the Euclidean distance of our feature, just mapped to a much higher dimensional space, based our feature finding algorithm.

Just because the new feature wasn’t part of *initial* feature set doesn’t mean that we can’t match that feature against those in a set that was previously computed. If that was the case, we wouldn’t be able to use online databases like Caltech-UCSD Birds, or INRIA Movie Actions.

4) **When is it more efficient to create an inverted file index to match a query image to other images in the database, rather than comparing the query to all database images without an index?**

There is an overhead cost to creating an inverted file index, but that cost is worthwhile when are going to be matching many (hundreds, thousands, or more) images against this index.

So, while the setup cost for creating the inverted index may be steep, it is more efficient in the long-term if we plan on using the inverted index data structure frequently, after it has been setup.

The inverted index will be ***much*** more efficient than querying against the whole database image by image. But if we only need to do one or two queries on the database, then overhead cost associated with created the inverted file index is not justified. It is probably faster to just do a small set of lookups on the whole DB, rather construct a complicated data structure that will only be used a handful of few times, total.

5) **Why do we need to measure both precision and recall in order to score the quality of retrieved results?**

Precision is a measure of *how accurately* we can **correctly identify** features, while recall is a measure of *how often* we can match features.

For an image matching algorithm to be useful, it needs to be 1) correct and accurate, and 2) able to give us wide enough variety of matches to help us find interesting data. Because of this, we need to measure and find a balance of precision vs. recall in our retrieval algorithms.

On the one extreme, it's possible that we can be **very precise**, but only willing to call something a match if we are extremely certain, thereby missing many true matches that don't quite make the cutoff, and rendering our data set too small to be useful.

On the other extreme, we could have **very high recall**, but then we would get too many false positives and our data set would be too inaccurate to be of much use.

Therefore, it is essential to find a good balance.

6) **Please write pseudocode for the `computeBOWRepr` function from HW5P.**

```
Start with:

    descriptors = Mx128 set of descriptors for the region

    means = 128xk set of cluster means for the image region

Action:

    bow = 128xk vector, each value set to zero for now

    For d in descriptors:

        index = cluster center that d is closest to, after calling dist2.m function

        word = cluster_centers[index]

        bow[index] += 1  // gives us frequency count

                         // for how many times this feature was matched

    return → norm(bow)  // normalize each entry in the bow vector, and return that matrix
```