K-means clustering: some tricks and tips

Initial centers: People initialize the centroids in a variety of different ways. Choosing random points will work. Or you can try the k-center-like approach we described where you try to generate centers at *k* points that are as far from each other as possible.

To simplify this initializing process, you could run it on a randomly selected fraction of the database. There's no need to consider every single point at this stage: a subset of $1/100^{th}$ of the database would work well enough.

Limiting dimensionality: In my experience, it's sufficient to use the top 5000 words in the lexicon. You could assess this by counting all occurrences of each word. But an even better way is to choose the words that appear in the greatest number of data points (pages or volumes). This tends to avoid words that may be very common in a small number of volumes.

At an earlier stage of this I said that we might need to treat words differently depending on whether they appear in "prose" or "verse" portions of a page, but I now think that's overkill for clustering. We can add wordcounts together, whether they appear in "-1" (prose) or "-2" (verse) portions of a page.

We'll also have three structural features created as ratios:

- numberOfLinesOnThisPage / meanNumberOfLinesPerPageInTheVolume
- numberOfCapitalizedLines / numberOfLinesOnThisPage
- thisPageNumber / totalNumberOfPagesInVolume

Iteration: Generally, in each iteration, data points are assigned individually to mappers that try to pair each data point with its nearest cluster center. The mappers output a key-value pair where the key is the cluster center identified as closest, and the value is the data point.

This allows points in the same cluster to be grouped by the reducer, which computes a new centroid for the cluster by simply taking the mean value of all the members.

It can happen that, at some stage of iteration, you get a cluster with no members. There are a variety of ways to handle this situation. You could reinitialize the cluster at a random point, or try to select the farthest point from existing cluster centers. But it

may actually be preferable to allow a cluster like this to "die." Perhaps our setting for k was too high. We'll need some way to indicate that a cluster has "expired" so mappers no longer try to measure the distance to it.

Distance metrics: What distance metric should we use in the mappers?

In clustering text documents, we rarely care about the absolute magnitude of a vector (which simply reflects the length of the text). This is one reason to prefer cosine similarity to Euclidean distance: the cosine similarity of two vectors remains unchanged if you multiply one of them by a constant. But I believe there are also other reasons to prefer cosine similarity, having to do with sparsity and "the curse of dimensionality."

Cosine similarity is basically the dot-product of two vectors divided by the product of their magnitudes. Of course, since similarity is the opposite of distance, the center that's closest to a given data point is going to be the one with the *highest* cosine similarity. If you prefer, you can calculate 1 - cosinesimilarity and treat that as a distance.

I'm also going to recommend that we use a normalization related to "Mahalanobis distance." The basic idea here is that we want to allow clusters to be ellipsoidal rather than circular/spherical. For a given cluster, some dimensions are really important (because the cluster distribution is narrow in that dimension), but other dimensions make little difference (the cluster spreads out widely in that dimension).

I think "Mahalanobis distance," technically, would only apply in a Euclidean space. But a simple hack can give us the same effect with an angular measure like cosine similarity. Basically, when we're calculating the distance to cluster center *c*, we can normalize both vectors by dividing each element of the vector by the standard-deviation of cluster *c* in that dimension. This will have the effect of reducing the importance of dimensions where there's already a lot of variance, and increasing the importance of dimensions where a given cluster is already tight.

In practice, we need to make this formula a little more complex to make it work. First of all, the place to calculate standard deviations is in the reducer. After calculating the new centroid *c* for a given cluster, we can calculate a vector of standard deviations this way:

```
For each point p in a cluster of n points:
      For each dimension d in the vector for this point:
            SumofSquaredDifferences[d] += (c_d - p_d)^2
            (in other words, we're creating a vector where each element
            d sums the squared differences between the centroid c and
            all points p in that dimension d)
Then for each dimension d, let's say
      StdDeviation[d] = sqrt(SumofSquaredDifferences[d] / n) + α
```

N is the number of points in the cluster. Alpha is a constant that we're adding because we don't want the standard deviation to become zero in any dimension. We would get a division by zero error later on, and also, we would be making the cluster infinitely narrow in that dimension. Initially, a value like 0.05 seems about right for alpha.

Then we need to normalize the vector of standard deviations to prevent clusters from growing or shrinking endlessly. In other words

```
For each dimension d,
      StdDeviation[d] = StdDeviation[d] / sum(StdDeviation)
```

This will guarantee that the vector sums to 1. After normalizing the `StdDeviation` vector in this way, we can save it in the same Java object that contains the cluster center for this cluster.

Then in the next iteration of the mapper, when we're measuring the distance (cosine similarity) to a particular cluster center, we can first divide each element of both vectors by the appropriate standard deviation for that cluster. This will reduce the importance of dimensions with big standard deviations, and increase the importance of dimensions where the cluster is narrow. (It will also, incidentally, tend to reduce the importance of very frequent words — which is okay.) We'll take the cosine similarity of these normalized vectors instead of the raw vectors; the cluster that has highest *normalized* cosine similarity will be considered "closest" to the data point.

Obviously, in the first iteration of the algorithm, we won't yet know the standard-deviation vectors for the clusters. We can initialize them all to 1, or some constant value.

I admit this is all a little hacky. It's a way of imitating the flexibility associated with model-based clustering (e.g., ellipsoidal clusters), while still keeping the simplicity of an algorithm that firmly assigns each point to a single cluster. (I think true model-based clustering might be a little trickier to parallelize with map-reduce, and also, I don't really want fuzzy clusters. I like the simple way k-means partitions the data.)

I'm getting some of the ideas here from this old doctoral thesis:
http://www.dtic.mil/dtic/tr/fulltext/u2/a321151.pdf

But I've also tested some of these ideas myself on a small scale, and they seem to work well in this domain.

We'll want to have an ability to export the vector of standard deviations, paired with the features (dimensions) they're associated with, so we can identify the particular features that turned out to be most important for defining a given cluster. (In some cases it may be a word like "table" or "contents"; in other cases it may be a structural feature like the ratio of pagenum/volpages.)