Clustering and textual data

1 General overview of clustering

Philipp and Miguel already introduced some techniques and algorithms, I would like now to give a more general overview. Why? First because I find it interesting, secondly because I would like to stress the point that there is no such thing as the Unique Correct Algorithm for clustering data (well, Philipp already showed some problems that arise with k-means or soft k-means, but as soon as you choose a different algorithm you can build counterexamples to it).

1.1 Notation.

We will now set some basic notation.

- A *pattern* or *observation* $\boldsymbol{x} = (x_1, ..., x_d)$ is a single data item; it is usually a vector; its elements are called *features*. The space where they are is called *feature* or *pattern space*, usually $\mathbb{Z}^d$ or $\mathbb{R}^d$.
- A *pattern set* is a set of patterns, which can be viewed as an $n \times d$ pattern matrix.
- A *class*, on the other hand, is something abstract that drives the pattern formation. Problem: how do we define concrete rules to mimic these classes?
- A *distance* or *similarity measure* is a metric or quasi-metric[1] on the feature space that (tries to) capture and quantify the similarity of patterns.

There are some degrees of freedom in choosing a clustering algorithms. We present here some possible classifications. Most of the (bi-)partitions showed here are orthogonal one to the other.

Hard and soft (or fuzzy) clustering. In the first approach patterns are univocally assigned to a cluster, while in the second one every pattern is labelled with some measure of its membership in every cluster (think probability). Usually fuzzy clustering is transformed to hard clustering after the last iteration of the algorithm, returning membership to the most likely cluster.

Deterministic and stochastic. This is relevant in the square error minimisation algorithms that we already saw. We can either proceed with standard deterministic techniques or, in particular when the dataset is very big, we can do a random search[2] on the space of all possible labelings.

Agglomerative and divisive. In the first case we start with each pattern belonging to its own cluster (a singleton) and then the algorithms joins clusters together, until a suitable stopping condition is met. In the second one we begin with a single cluster which is then split over and over, until the stopping condition is met.

Monothetic and polythetic. In this direction we can either consider one feature at a time or more than one or all at the same time while computing the similarity measures.

1. A quasi-metric is a non necessarily commutative metric.
2. In some sense, there is a lot of room to make choices here.

The tree.

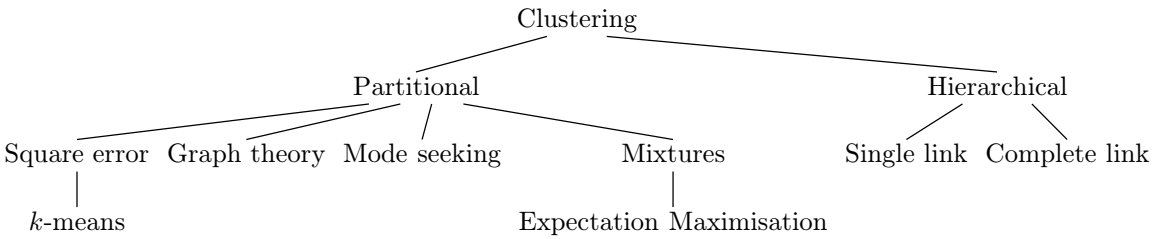

Figure 1. Partial taxonomy of clustering algorithms

Complexity of some algorithms. Just to get an idea. We have n number of patterns considered, k number of clusters, l number of iterations for the algorithm to converge.

Clustering algorithm	Time Complexity	Space complexity
leader	$O(k\,n)$	$O(k)$
k-means	$O(k\,n\,l)$	$O(k)$
Shortest spanning path	$O(n^2)$	$O(n)$
single-link	$O(n^2 \log n)$ or $O(n^2)$	$O(n^2)$
complete-link	$O(n^2 \log n)$	$O(n^2)$

Table 1. Complexity

1.2 Pattern representation, feature selection and extraction.

We start with data that usually is not a point in a numerical space. Think apples and bananas. So what do we do? We select and extract some features that we believe are most informative about our data. The we have to represent them in the feature space. This might sound trivial, but is actually extremely important: if we have a crescent shaped set of data and we represent it in polar coordinates we are probably going to see it as a cluster, while if we represent it in cartesian coordinates that might not happen. Moreover depending on the problem we are trying to solve, our features could be obvious or very obscure, which is particularly true when the pattern we are trying to capture is not so quantitatively well defined (the digits in the challenge proposed by Miguel: what are the features that capture the essence of "seven-ness"?). As is is often the case for data, we have the following general structure:

- Quantitative: continuous or discrete.
- Qualitative: ordinal or nominal.

There are several techniques we could use to select those features that are really relevant, as mentioned already the most common ones (excluding trial and error) are principal component analysis (PCA) and expert elicitation (in this case meaning: you find someone who knows about this kind of problem or phenomenon and you ask her which are the features that discriminate more).

1.3 Similarity measures.

Usually standard Minkowski metrics (in most cases Euclidean distance), but other are possible, in particular some are quite obvious in human perception terms, but are more difficult to render mathematically (conceptual similarity measures). The choice of the distance is a model choice, like it was for the feature selection, the pattern representation and the feature extraction. This means that there is no correct recipe, but it falls on the shoulders of the data scientist. Also the algorithm choice and the cluster representation that we'll see in the following sections are model choices. Of course, you could program a machine to choose the proper clustering approach given a problem or a dataset: but that is already very close (IMO) to AI.

1.4 Cluster representation.

Sometimes we are just looking for a partition of the data we have, so we are interested in the whole composition of each cluster. Other times (for example when we have ten of thousands of data) we might look for a more compact description of the clusters. This is particularly true when we are doing exploratory analysis, looking for some underlying class that we have not an a priori abstract description for. This representation is sometimes called *abstraction*. It plays a very important role: it allows for a more compact and understandable description, hence improving the chances for an effective data compression and the efficiency of decision making. Let's see some possible strategies.

- Describe the cluster using the centroid. Most common, but it does not work well for non isotropic and sparse clusters.
- Describe the cluster using a set of boundary points. Allows us to capture more complex shaped clusters, but we have to decide a number of points.
- Describe the cluster through a classification tree or though logical expressions. This approach is easily understood, bu tas a severe limitation: it works only for rectangular or isotropic clusters. On the other hand it is perfectly shaped for binary search, having already a binary tree structure.

2 Text clustering

Motivation. The study of clustering started with quantitative data, mostly because quantitative data is often available from lots of different sources and it makes more sense to begin with broad spectrum problems, leaving specialisation to a second phase. At the same time there are obvious advantages in applying clustering to text data: classification, organisation and browsing of documents or summarisation of corpus. We focus for the moment on unsupervised learning, but of course the classification part (which turns out to be extremely interesting) has some intrinsic difficulties, as the algorithm has to learn when documents are related. That provides also some interesting insight in the way we establish our own critical thinking: how do you determine whether two documents *in a specific set* are related or not?

2.1 Example: classification of Wikipedia pages.

Let's say we have a collection of Wikipedia pages and we want them clustered in some way. This will provide us with a case study in which to show the steps mentioned above.

Feature selection. How do we characterize documents that are similar? One first strategy might be to just count the number of words they have in common. This is not a completely wrong idea, but on second thought we realise that it has some serious drawbacks: there is a lot of noise, meaning that there are a lot of words that are very likely to be in any document we consider (e.g. "the", "is", and so on). So a first step to clean data might be the removal of the so-called *stopwords*.

We keep focused on the common words strategy, but after we remove the stopwords we still have bad situations, for example a very long document is more likely to have words in common with lots of different other documents, and just checking the presence or not of a specific term does not provide a lot of information. The word might even have a different meaning! Moreover all the words are equal, but some are more equal than the others. The less frequent the words are the more important. This is a natural step following the idea of stopwords. So, how do we take advantage of this idea?

The plan is to introduce the **tf-idf** measure, which stays for *term frequency, inverse document frequency*. Every word gets for every document a number measuring its relevance in the document with respect to the whole set of documents (or corpus). The measure is computed in the following way: the *term frequency* is literally the frequency of the word in the document[3],

$$\mathrm{tf} = \frac{\#\{\text{occurrences of specific word in document}\}}{\#\{\text{words in document}\}}.$$

The *inverse document frequency* is the logarithm of the inverse frequency *sensu stricto*:

$$\text{idf} = \log\left(\frac{\#\{\text{documents in corpus}\}}{\#\{\text{documents with the specific word}\}}\right).$$

The two are then multiplied.

Similarity measure. Now we have a vector space where the documents are represented as vectors, in which each feature is a word, represented through its tf-idf measure. How do we compute the distance or (dis)similarity between documents? We compute the angle between them, through the cosine. We have, given two documents d_1 and d_2 represented in the vector space just defined, that their cosine is:

$$\cos(d_1, d_2) = \frac{d_1 \bullet d_2}{|d_1|\,|d_2|},$$

so their angle is just the arccosine of this quantity. The further that two documents can be is $\pi/2$, because coordinates are all positive.

3 Some clustering algorithms

3.1 Hierarchical agglomerative clustering.

i. Compute proximity matrix with the distances between all pairs of patterns (recall that we start with every pattern being a cluster).

ii. Search the proximity matrix for the closest clusters, merge them and update the proximity matrix.

iii. If we are down to one cluster, STOP, otherwise GOTO ii.

The complexity of HAC is $O(n^2 \log n)$: the computation of similarities is $O(n^2)$, and with suitable techniques (priority queues, for those interested) the search and merge can be done in $O(n^2 \log n)$, hence the complexity.

3.1.1 Simple-link.

In this specific case of HAC we measure the similarity of two cluster in the best case, that is considering the two most similar elements. In other words we just look at the place where two clusters are closer one to the other:

$$d_{\text{sl}}(C_1, C_2) = \min_{d_i \in C_i} d(d_1, d_2).$$

Because of its definition this algorithm is great in finding chain structures (which might be good or bad). In this case, as hinted by the table, we can slightly improve the complexity of the algorithm, if we keep track of which is the best merge for each cluster, lowering complexity to $O(n^2)$. This work only for simple link, as we use the minimum distance, so the best merge is preserved after every iteration (up to the computation of a max , which is only $O(n)$ complex).

3.1.2 Complete-link.

Another take on HAC is to consider the most dissimilar points in the clusters, that is the worst case:

$$d_{\text{cl}}(C_1, C_2) = \max_{d_i \in C_i} d(d_1, d_2).$$

3. To be completely honest there are other possible definitions of term frequency.

Now we choose as the next cluster the one with the smallest diameter, taking into account a more global picture of the cluster. This algorithm does not focus on chains, but focuses on compact clusters. One big problem is that it is extremely sensitive to outliers. Fun fact: there is an efficient version of this algorithm, called CLINK, which has a complexity of $O(n^2)$, but has the huge drawback of giving unsatisfactory results in information retrieval due to a not exact hierarchy. The $O(n^2)$ is the optimal complexity bound.

3.1.3 Average-link.

Here we have a shift, compared to the previous two examples: instead of comparing two clusters considering only a pair of patterns, we consider the whole set of patterns involved, that is we consider *all* pair of patterns, including patterns already in the same cluster (but not self-similarities). Let's write down the distance:

$$d_{\text{av}}(C_1, C_2) = \frac{1}{(n_1+n_2)(n_1+n_2-1)} \sum_{p_n \neq p_m \in C_1 \cup C_2} p_n \cdot p_m$$

and this can be computed efficiently (as long as we represent our patterns in $\mathbb{R}^d$), since we can rewrite the double sum thanks to the distributivity of sum with respect to product, hence:

$$d_{\text{av}}(C_1, C_2) = \frac{1}{(n_1+n_2)(n_1+n_2-1)} \left(\left(\sum_{p_n \in C_1 \cup C_2} p_n \right)^2 - (n_1+n_2) \right).$$

Here we must stress that there are a couple of differences with the general algorithm:

1. we must have patterns represented as $\mathbb{R}^d$ vectors;
2. the vectors have to be normalised so that self-similarities are 1;
3. we have to use the dot product and not other distances.

One last remark: the reason we remove self-similarities is that keeping them means penalise the larger clusters in comparison to the smaller ones (due to the lower proportion of self similar distances).

3.1.4 Centroid-link.

Here the similarity is defined also in term of two points, the centroids, but these are computed from all points of each cluster, so it does not show the problems that single- and complete-link had. Alas, it has a much bigger problem: it is not monotone! In all previous hierarchical agglomerative algorithms the similarity between clusters *decreases* in time (that is the more merges, the less similarity between things merged); in the centroid-link we lose that property. The reason this algorithm has been considered is that it has an $O(n^2)$ complexity in time.

3.1.5 Cutting the dendrogram.

With hierarchical algorithms what we get is not a partition, but a dendrogram. We can obtain a partition from the dendrogram, but in order to do so we need to cut the dendrogram. How do we do that? There are several possible choices (also model choices).

- Lazy: cut at some pre-specified level l of similarity. That is to say, we don't want clusters with a combined similarity less than l.
- Lazy 2: as in the k-means clustering presented by Philipp[4] we fix the number of clusters we want to end up with.

4. I am not implying that Philipp is lazy, as the opposite is definitely true as anyone can witness.

– Largest gap: we cut the dendrogram right before the largest gap, which is the one that "lowers" the most the quality of the clustering. Actually this "rule of thumb" is the most used whenever there are dendrograms involved in statistics and follows the same idea as the scree plot rule that we have for factor analysis.

– RSS: we (ab)use the following formula

$$k = \text{argmin}_h(\text{RSS}(h) - \lambda\, h),$$

where h is the cut resulting in h clusters, λ is the penalty for each additional cluster and RSS is the residual sum of squares (or any other distortion measure).

3.2 Partitional clustering (distance based).

We briefly sketch the ideas behind some partitional algorithms for data clustering, focusing on their application to text data.

3.2.1 *k*-anchored.

We start by choosing some representative documents (pattern) from our set of data, called anchors: around those points we build a first batch of clusters, assigning each document to the cluster determined by the closest anchor. The underlying idea is to find an optimal set of documents to play as anchors. We compute the average similarity of each document to its closest anchor and this is the utility function that we want to maximise. In each iteration we replace an anchor picked at random with another randomly picked document from the whole set of data if it increases the utility function. As is the case with other iterative algorithms we proceed until convergence is obtained.

There are disadvantages to this approach.

– We have to recompute the utility function at each iteration and we need a large number of iteration to achieve convergence, so the algorithm might be quite slow. The complexity is $O(k\,n\,l)$.

– Moreover k-anchors do not work well with sparse data, and text data is very sparse. Indeed, many documents (once we remove common words) have little to none words in common. This might become a huge problem.

3.2.2 *k*-means

We know this algorithm well, so we'll just see some comments about its use in text data clustering. One advantage of this algorithm is how quickly it converges, compared to k-anchors: for large text data sets al little as 5 iterations are enough to converge. We have disadvantages, of course, the first being that the algorithm is very sensitive to the initial set of clustering seeds. Another drawback is that the centroid of a text data cluster will contain a large number of words, so the computation of similarity measures is slower.

Bonus question: What is the centroid of a set of documents?

3.3 Scatter-gather algorithm, an hybrid.

With this algorithm we try to take advantage of the strong suits of both hierarchical and partitional approaches. We start by using a hierarchical clustering on a sample of the whole set of documents to find a good set of seeds, which we'll provide to a k-means algorithm afterwards, to cluster the whole set of documents.

3.3.1 Creating the set of seeds.

There are two main algorithms used to create the starting set of seeds.

- Buckshot. Instead of just picking k random seeds, we choose $\sqrt{n\,k}$ points and then we use a hierarchical algorithm to cluster them in k clusters. Complexity is $O(n \cdot k)$, as we don't need to get a *perfect* hierarchical structure, but only a good one (so we can use one of those $O(n^2)$ version of the algorithms).

- Fractionation. We split the whole set into n/m "buckets" of size $m > k$. We use a hierarchical algorithm to each bucket to shrink it by a factor $\lambda < 1$. We now have $\lambda \cdot m$ clusters in each bucket, which we can agglomerate in a single document just by merging all members. Overall we have now $\lambda\, n = n/m \cdot (\lambda\, m)$ documents. Now we can repeat the process on the resulting documents, until we are left with k seeds. What about the complexity? The hierarchical algorithm costs (at the first iteration) $O(m^2)$ for each bucket, so $O(n \cdot m)$ overall. On the following iterations it costs $O(\lambda^j \cdot n \cdot m)$ overall, so for $\lambda < 1$ we still get a $O(n \cdot m)$ total complexity. So we can pick $m = O(k)$ and get a total complexity of $O(n \cdot k)$.

3.3.2 Clustering through *k*-means.

Up to this moment the complexity is equivalent in both cases to one iteration of the k-means algorithm. Now we run the algorithm itself with the starting seeds so defined, getting an overall complexity in time of $O(n \cdot k \cdot l)$, but avoiding the issues of k-means.

3.3.3 Possible improvements.

There are techniques that refine even more the clustering obtained, at no additional complexity cost. We just cite them here: split and join.

3.4 Word based clustering.

There are other approaches designed specifically for texts, which are not just (high dimensional) vectors of words, but have a lot of underlying structure. Words go often together, so we can look for *word patterns* instead of just words or even have a two steps clustering, clustering first words within a document and then documents, seen as vectors of word clusters. Another approach along this direction is the simultaneous clustering of words and documents (co-clustering) using information-theoretical approaches or graph theoretical ones. Finally let's mention frequent phrases clustering, where we consider not one word at a time or an unordered set of words, but we focus on actual, concrete phrases.

3.5 Topic models.

This is just a sneak peek on the shape of things to come: we can see the set of documents as a function of hidden random variables, and we estimate the parameters of such RVs using a selected collection of documents. In particular we'll assume that each document has a vector of probabilities of belonging to one of the k clusters (topics, in this framework) and also every topic has an associated probability vector, made of the probabilities of each term in the lexicon for the topic.