[MITx 6.86x Notes Index]

Unit 04 - Unsupervised Learning

Lecture 13. Clustering 1

13.1. Unit 4: Unsupervised Learning

In this unit we start talking of problems where we don't have labels, where we need to find a structure in the data itself. For example, Googe news provide clusters of stories over the same event. It introduce a structure in the feed of news stories.

We start looking at hard assignment clustering algorithms, where we assume that each data point can belong to just a single cluster of (somehow to define) "similar" points.

We will then move to recognise data structures where each point can belong to multiple clusters, trying to discover the different components embedded in the data and looking at the likelihood a certain point was generated by that component.

And we will introduce generative model, which provide us the probability distribution over the points. We will look at multinomials, Gaussian and mixture of Gaussians. And by the end of this unit, we will get to a very powerful unsupervised learning algorithm for uncovering then, the line structure, which is called the expectation–maximization (EM) algorithm.

We will finally practice both clustering and the EM algorithm in a Netflix-recommandation project in Project 4.

13.2. Objectives

- Understand the definition of clustering
- Understand clustering cost with different similarity measures
- Understand the K-means algorithm

13.3. Introduction to Clustering

The topic in this unit is unsupervised machine-learning algorithms. In unsupervised learning, we will still have a training set, but we will not have a label like in supervised learning.

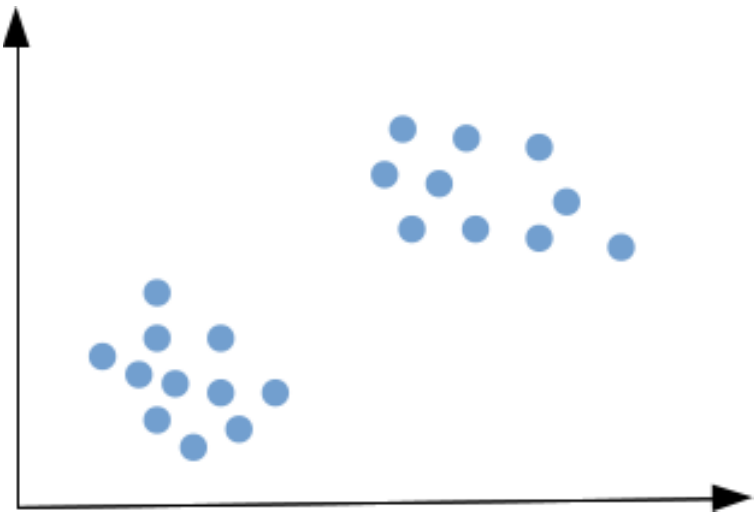
So in this case, we are provided with just a set of feature vectors, and we are

trying to learn some meaningful structure which would represent this set. In this lesson in particolar we will cover *clustering* algorithms.

We will first start from the definition of a cluster.

While in a *classification* problem we already have the possible categories and the label associated to it in the training data, in clustering we need to find a structure in the data and associate it with features of the data itself, in order to predict the category from the data itself without the need for a explicit label.

For example take this chart:



It is intuitive that the points belong to two different categories, in this case based on their position on the (x,y) axis, even if the points have no label (colour) associated to them.

Similarly, in many cases the meaningful structure is actually quite intuitive in the context of a specific application.

Find a structure is however not enough. We need to be able to describe how the structure we found is good in representing the data, how good is the cluster, the partitioning of the data (the “clustering cost”). And we will need for that a measure of similarity between points.

We will finish the lecture dealing with the K-means algorithm, an important and widely used algorithm which actually can take any set of training point and find a partitioning to K clusters.

To conclude the segment with an example, let’s consider again Google news. In google News we have both an example of classification and one of clustering.

The grouping of the news in one of the categories (sport, science, business,…) is a categorisation task. The categories are fixed, and someone has trained the algorithm manually setting the categories of each news for a training set, so that now it can be predicted automatically.

The grouping of news around individual new events (the covid-19, the brexit, the US elections…) is instead a clustering problem: here the categories are not

predefined and there has been no training with manually label all possible events.

What do we need for such task?

- We first need to provide a representation of the news stories, for example employing a *bag-of-words* approach, giving a (sparse) vector representation to each word;
- Second, we need to decide how to compare each of these representations, in this case vectors;
- Finally we need to implement the clustering algorithm itself, based on the pairwise measure of proximity found in step two, or, for example, selecting one representative story to show as a central story in the news.

13.4. Another Clustering Example: Image Quantization

If the previous example was dealing with text clustering for visualisation purposes, let's now deal with image clustering for compression purposes, and more specifically for creating a colour palette for the image, a form of **image quantisation** ("quantisation" refers to the process of mapping input values from a large setto output values in a countable smaller set, often with a finite number of elements).

A typical 1024 x 1024 pixel image coded in 24 bits per pixel (8 bits per each of the RGB colours) would size 1024x1024x24 = 25165824 bits = 3 MB.

If we create a palette restricted to only 32 colours, and represent each pixel with one of the colours in this palette, the image then would size only 1024x1024x5+5*24 = 5243000 bits = 640.01 KB (this by the way is a compression technique used in PNG and GIF images).

Note that the image quality depends from the number of colours of the palette we choose, i.e. the number of clusters K. While cartoon, screenshots, logos would likely require very small K to appear of a quality similar to the original image, for photos we would likely need higher K to achieve the same quality.

Still, to get the compressed image we need to undergo the three steps described in the previous segment. We need to represent the image (as a vector of pixels with a colour code). The second step is to find some similarity measure between colours, and finally we can run the clustering algorithm to replace each pixel 24 bits representation with the new 5 bits one, choosing a map from the 12777216 original colours (24 bits) to the 32 "representative ones" (also to be found by the algorithm!).

13.5. Clustering Definition

There are two concepts related to a cluster. The first one is the partitioning of the data, how the different data records are distributed among the clusters. The second one is the representativeness, how we aggregate the cluster members in a representative element per each cluster.

Partitioning

Let's $S_n = \{x^{(i)} | i = 1, \ldots, n\}$ be the set of n feature vectors indexed by i and $C_K = \{C_j | j = 1, \ldots, K\}$ being the set of K clusters containing the *indexes* of the data points, then $C_1, \ldots, C_j, \ldots, C_K$ forms a K-partition of the data if, at the same time, their union contain all the data indexes and their intersection is empty: $\cup(C_1, \ldots, C_j, \ldots, C_K) = \{1, 2, \ldots, n\}$ and $\cap(C_1, \ldots, C_j, \ldots, C_K) = \{\}$

This partitioning define a so-called **hard clustering**, where every element just belongs to one cluster.

Representativeness

The representativeness view of clustering is the problem of how to select the feature vector to be used as representative of the cluster, like the new story to put as header on each event in Google News or the colour to be use for each palette item. Note that the representative element for each cluster doesn't necessarily be one existing element of the cluster set, can be also a combination of them, like the average.

We will see later in this lesson what is the connection between these two views. Clearly they are connected, because if you know what is your partitioning, you may be able to guess who is the representative. If you know who is a representative, you may be able to guess who is a constituent, but we will see how we can actually unify these two views together.

Note that finding the optimal number of K is itself another problem, as K is not part of the clustering algorithm, it is an input (parameter) to it. That is, K is a hyperparameter of the clustering algorithm.

13.6. Similarity Measures-Cost functions

Given the same set of feature vectors, we can have multiple clustering results, multiple way to partition the set. We need to define a "cost" to each possible partition of our data set, in order to rank them (and find the one that minimise the cost). We will make the assumption that the cost of a given partition is the sum of the cost of its individual clusters, where we can intuitively associate the cost to some measure of the cluster heterogeneity, like (a) the cluster diameter (the distance between the most extreme feature vectors, i.e. the outliers), (b) the average distance or © (what we will use) the sum of the distances between every member and z_j, the representative vector of cluster C_j.

In turn we have to choose which metric we use to measure such distance. We have several approaches, among which:

- cosine distance, $dist(x^{(i)}, x^{(j)}) = \frac{x^{(i)} \cdot x^{(j)}}{||x^{(i)}|| * ||x^{(j)}||}$
- square euclidean distance, $dist(x^{(i)}, x^{(j)}) = ||x^{(i)} - x^{(j)}||^2$

The cosine distance has the merit of being invariant to the magnitude of the vectors. For example, in terms of the Google News application, if we choose cosine to compare between two vectors representing two different stories, this

measurement will not take it into account how long are the stories. On the other hand, the Euclidean distance would be sensitive to the lengths of the story.

While cosine looks at the angle between vectors (thus not taking into regard their weight or magnitude), euclidean distance is similar to using a ruler to actually measure the distance (and so cosine is essentially the same as euclidean on normalized data)

Cosine similarity is generally used as a metric for measuring distance when the magnitude of the vectors does not matter. This happens for example when working with text data represented by word counts. We could assume that when a word (e.g. `science`) occurs more frequent in document 1 than it does in document 2, that document 1 is more related to the topic of science. However, it could also be the case that we are working with documents of uneven lengths (Wikipedia articles for example). Then, `science` probably occurred more in document 1 just because it was way longer than document 2. Cosine similarity corrects for this (from https://cmry.github.io/notes/euclidean-v-cosine).

While we'll need to understand which metric is most suited for a specific application (and where it matters in the first instance), for now we will employ the Euclidean distance.

Given these two choices, our cost function for a specific partition becomes:

$$cost(C_1, \dots, C_j, \dots C_Z, z^{(1)}, \dots, z^{(j)}, \dots, z^{(Z)}) = \sum_{j=1}^{Z} \sum_{i \in C_j} ||x^{(i)} - z^{(j)}||^2$$

Note that for now we are giving this cost function as a function of both the partition and the representative vectors, but when we will determine how to compute the representative vectors z_j endogenously, this will be a function of the partition alone.

13.7. The K-Means Algorithm: The Big Picture

How to find the specific partition that minimise the cost ? We can't iterate over all partitions, measure the cost and select the smaller one, as the number of (non empty) k-partitions of a set of n elements is huge.

To be exact it is known as the "Stirling numbers of the second kind" and it is equal to:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

For example, $\left\{ {3 \atop 2} \right\} = 3$, but $\left\{ {100 \atop 3} \right\} \approx 8.58e+46$

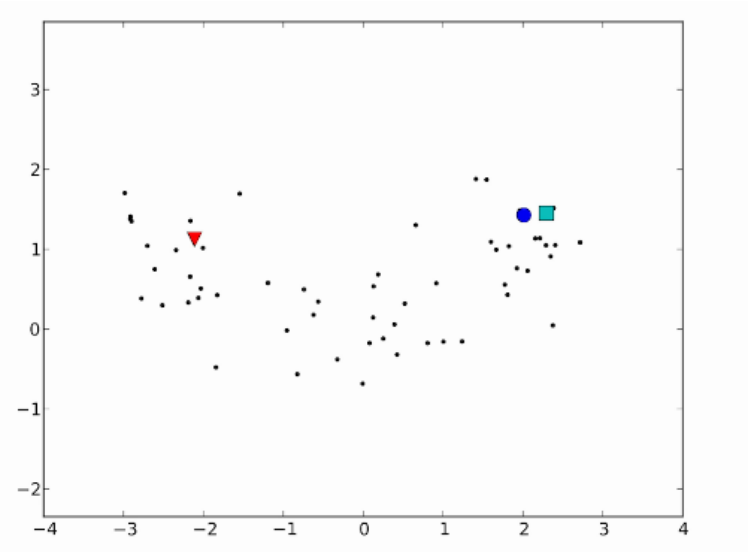
The K-M algorithm helps in finding the minimal distance in a computationally feasible way.

In a nutshell, it involves (a) to first randomly select k representative points. Then (b) iterate for each point to assign the point to the cluster of the closest representative (according with the adopted metric), and (c) move each

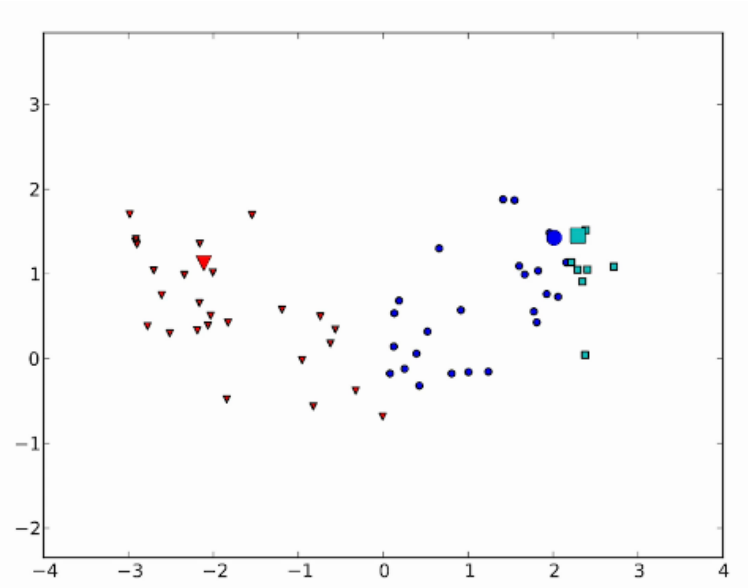
representative at the center of its newly acquired cluster (where “center” depends again from the metric). Steps (b) and (c) are reiterated until the algorithm converge, i.e. the tentative k representative points (and their relative clusters) don’t move any more.

Graphically:

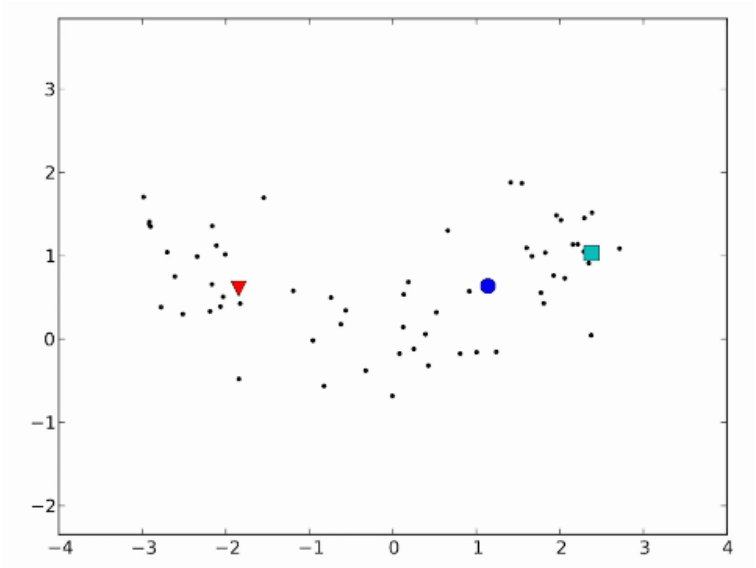
Random selection of 3 representative points in 2D:



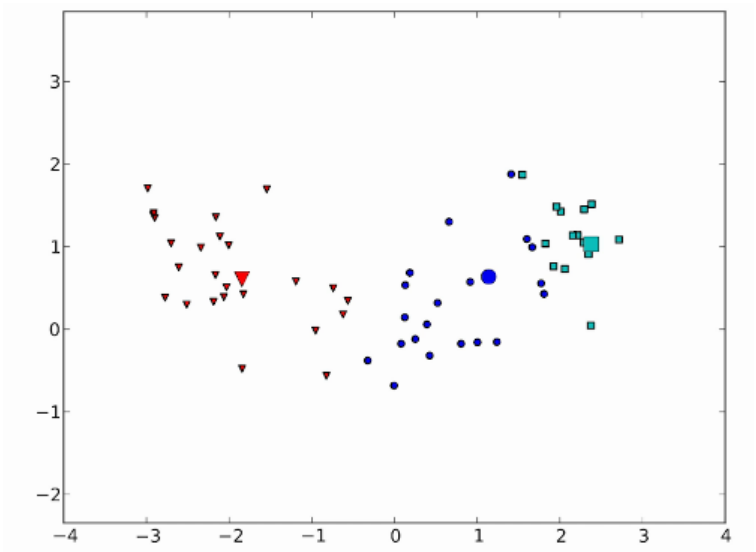
Assignment of the “constituent” of each representative:



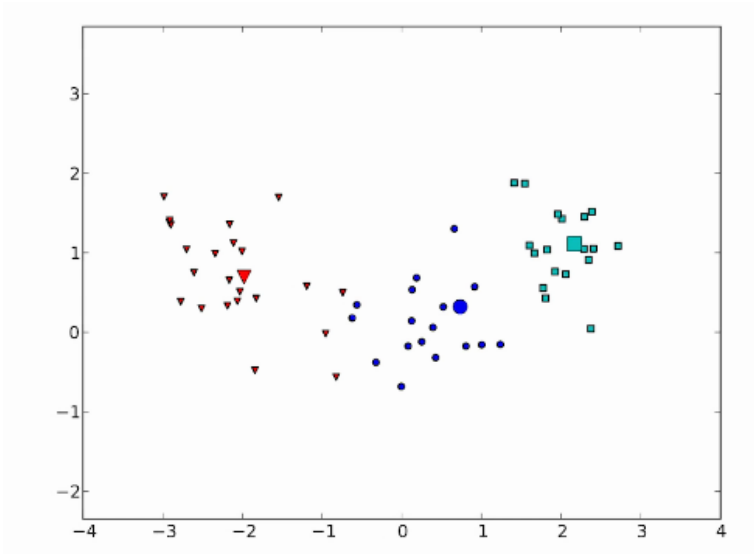
Moving the representative at the center of (the previous step) constituency:



Redefinition of the constituencies (note that some points have changed their representative):



Moving again the representatives and redefinition of the constituencies:



Let’s describe this process a bit more formally in terms of the cluster costs.

- 1. Randomly select the representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(K)}$
- 2. Iterate:
  - 2.1. Given $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, assign each data point $x^{(i)}$ to the closest representative $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, so that the resulting cost will be $cost(z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}) = \sum_{i=1}^{n} min_{j=1,\ldots,K} ||x^{(i)} - z^{(j)}||^2$
  - 2.2. Given partition $C_1, \ldots, C_j, \ldots, C_K$, find the best representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$ such to minimise the total cluster cost, where now the cost is driven by the clusters:
    $cost(C_1, .., C_j, \ldots, C_K) = min_{z^{(1)},\ldots,z^{(j)},\ldots,z^{(Z)}} \sum_{j=1}^{K} \sum_{i \in C_J} ||x^{(i)} - z^{(j)}||^2$

Note that in both step 1 and step 2.2 we are not restricted that the initial and final representatives are part of the data sets. The fact that the final representatives will be guarantee to be part of the dataset is instead one of the advantages of the K-medoids algorithm presented in the next Lesson.

13.8. The K-Means Algorithm: The Specifics

Finding the best representatives

While for step (2.1) we can just iterate for each point and each representative to find the representative for each point that minimise the cost, we still need to define how to do exactly the step (2.2). We will see later extensions to the KM algorithm with other distance metrics, but for now let's stuck with the squared geometric distance and note that each cluster select its own representative independently.

Using the squared Euclidean distance, the "new" z_j representative vector for each cluster, must satisfy $j : min_{z_j} cost(z_j; C_j) = min_{z_j} \sum_{i \in C_J} ||x^{(i)} - z_j||^2$.

When we compute the gradient of the cost function with respect to z_j and set it to zero, we retrieve the optimal z_j as $z_j = \frac{\sum_{i \in C_J} x^{(i)}}{|C_J|}$, where $|C_J|$ is the number of elements of the cluster C_J. Intuitively the optimal representative vector is at the center of the cluster, i.e. it is the centroid of the group.

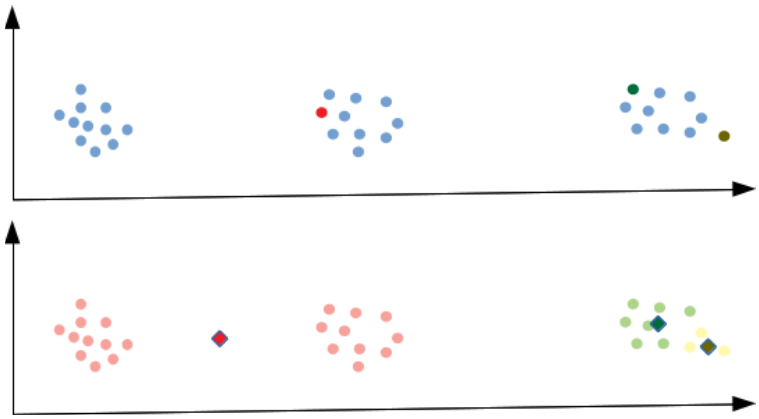
We stress however that this solution is linked to the specific definition of distance used, the squared Euclidean one.

Impact of initialisation

Note that the KM algorithm is guarantee to converge and find a *local* cost minimisation, because at each iteration the cost can only decrease, the cost function is non-negative and the number of possible partitions, however large, is finite.

It *doesn't* however guarantee to find a *global* cost minimisation, and it is indeed very sensitive to the choice of the initial representative vectors. If we start with a different initialization, we may get a very different partitioning.

Take the case of the above picture:



In the upper part we see the initial random assignation of the representative vectors, and in the bottom picture we see the final assignment of the clusters. While the feature vectors moved a bit, the final partition is clearly not the optimal one (where the representative vectors would be at the center of the three groups).

In particular, we may run into troubles when the initial representative vectors are close to each others rather than spread up across the multidimensional space.

While there are improvements to the K_Mean algorithm to perform a better initialisation than a random one, that take this consideration into account, we will use in class a vanilla KM algorithm with simple random initialisation. An example of a complete such KM algorithm in Julia can be found on https://github.com/sylvaticus/lmlj/blob/master/km.jl

Other drawbacks of K-M algorithm

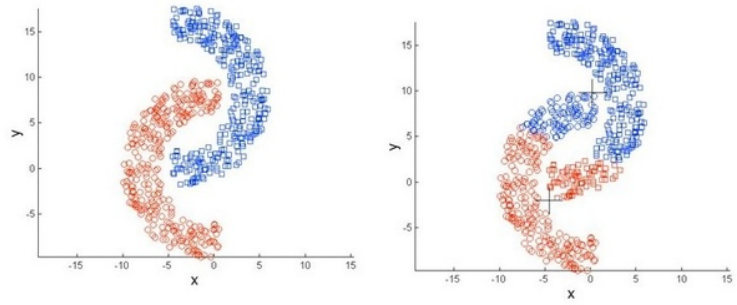
In general we can say there are two classes of drawbacks of the K-M algorithm.

The first class is that the measure doesn't describe what we consider as a natural classification, so the cost function doesn't represent what we would like the cost of the partition to be, it does not return a useful information concerning the partition ranking.

The second class of problems involves instead the computational aspects to reach the minimum of this cost, like the ability to find a global minimum.

While K-M algorithm scale well to large datasets, it has many other drawbacks.

One is that vanilla K-M algorithm tries to find spherical clusters in the data, even when the groups have other spatial grouping:



To account for this, k-means can be kernelized, where separation of arbitrary shapes can be reached theoretically using higher dimensional spaces. Or there could be used a regularized gaussian mixture model, like in this paper or in these slides.

Other drawbacks, includes the so called “curse of dimensionality”, where k-means algorithm becomes less effective at distinguishing between examples, the manual choice of the number of clusters (that need to be solved using cross-validation), the fact of not being robust to outliers.

These further drawbacks are discussed, for example, here, here or here.

An implementation of K-Means algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L65

Lecture 14. Clustering 2

14.1. Clustering Lecture 2

Objectives:

- Understand the limitations of the K-Means algorithm
- Understand how K-Medoids algorithm is different from the K-Means algorithm
- Understand the computational complexity of the K-Means and the K-Medoids algorithms
- Understand the importance of choosing the right number of clusters
- Understand elements that can be supervised in unsupervised learning

14.2. Limitations of the K Means Algorithm

On top of the limitations already described in segment 13.8 (computational problem to reach the minimum and the cost function not really measuring what we want) there is further significant limitation of the K-Mean algorithm: the fact that the z’s are actually not guaranteed to be the members of the original set of points x.

In some applications this is not a concern, but it is for others. For example, looking at Google News, if we create a representative of the cluster of the story

which doesn't correspond to any story, we actually have nothing to show.

We will now introduce the K-medoids algorithm that modifies the k-means one to consider any kind of distance (not only the squared Euclidean one) and return a set of representative vectors that are always part of the original data set.

We will finish the lesson discussing how to choose K, the number of K.

14.3. Introduction to the K-Medoids Algorithm

In K-means, whenever we randomly selected the initial representative points, we were not constrained to select within the data set points, we could select any point on the plane.

In K-Medoids algorithm instead we start by selecting the representatives only from within the data points.

The step 2.1 (determining the constituencies of the representatives) remains the same, while in step 2.2, instead of choosing the new representatives as centroids, we constrain again that these have to be one of the point of the cluster. This allow to just loop over all the points of the cluster to see which minimise the cluster cost. And as in both step 2.1 and 2.2 we explicitly use a "distance" function we can employ any kind of distance definition we want, i.e. we are no longer restricted to use the squared Euclidean one.

The algorithm becomes then:

- 1. Randomly select the representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(K)}$ from within $X^{(1)}, \ldots, X^{(j)}, \ldots, X^{(n)}$
- 2. Iterate:
  - 2.1. Given $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, assign each data point $x^{(i)}$ to the closest representative $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, so that the resulting cost will be $cost(z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}) = \sum_{i=1}^{n} min_{j=1,\ldots,K} ||x^{(i)} - z^{(j)}||^2$
  - 2.2. Given partition $C_1, \ldots, C_j, \ldots, C_K$, find the best representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$ from within $X^{(1)}, \ldots, X^{(j)}, \ldots, X^{(n)}$ such to minimise the total cluster cost, where now the cost is driven by the clusters: $cost(C_1, .., C_j, \ldots, C_K) = \sum_{j=1}^{K} min_{z^{(j)}} \sum_{i \in C_J} dist(x^{(i)} - z^{(j)})$

14.4. Computational Complexity of K-Means and K-Medoids

The Big-O notation

Let't now compare the computational complexity of the two algorithms, using the capital O (or "Big-O") notation, which talks to us about the order of growth, which means that we are going to look at the asymptotic growth and eliminate all the constants.

We often describe computational complexity using the "Big-O" notation. For

example, if the number of steps involved is $5n^2 + n + 1$, then we say it is “of order n^2" and denote this by $O(n^2)$. When n is large, the highest order term $5n^2$ dominates and we drop the scaling constant 5.

More formally, a function $f(n)$ is of order $g(n)$, and we write $f(n) \sim O(g(n))$, if there exists a constant C such that $f(n) < Cg(n)$ as n grows large.

In other words, the function f does not grow faster than the function g as n grows large.

The big-O notation can be used also when there are more input variables. For example, in this problem, the number of steps necessary to complete one iteration depends on the number of data points n, the number of clusters K, the dimension d of each vector x_i. Hence, the number of steps required are of $O(g(n, K, d))$ for some function $g(n, K, d)$.

The computational complexity of the two algorithms

We don’t know how many iterations the algorithms take, so let’s compare only the complexity of a single iteration.

The order of computation for the step 2.1 of the K-Mean algorithm is $O(n * k * d)$ as we need to go trough each point (n), compute the distance with each representative (k) and account that we deal with multidimensional vectors (d). The step 2.2 is the same, but because we’re talking about the asymptotic growth, whenever we sum them up, we’re still staying within the same order of complexity.

For the K-Medoids algorithm instead we can see that is much more complex, as in the step 2.2 we need to go trough all the clusters, and then all the point. Depending how the data is distributed across the cluster, we can go from the best case of all points in one cluster, and then we would have a computational complexity of $O(n^2 * d)$ to a situation where the points are homogeneously distributed across the clusters, and in such case we would have n/k points in each cluster and a total complexity $O(\frac{n}{k} * \frac{n}{k} * k * d) = O(\frac{n^2}{k} * d)$.

Given that normally $n \gg k$, what makes the difference is the n^2 term, and even in the best scenario the computational complexity of the K-Medoids algorithm is much higher than those of the K-Mean one, so for some applications with big datasets the K-Mean algorithm would be preferable.

At this point we already have seen two clustering algorithms, but there are hundreds of them available for our use. Each one has different strengths and weaknesses, and whenever we are selecting a clustering algorithm which fits for our application, we should account for all of them in order to find the best clustering algorithm for our needs.

14.5. Determining the Number of Clusters

First, let’s recognise that more K we add to a model, more the cost function decreases, as the distances from each point and the closer representative will

decrease, until the degenerated case where the number of clusters equals the number of data and the cost is zero.

When is it time then to stop the number of clusters ? To decide which is the “optimal” number K ? There are three general settings. In the first one the number of clusters is fixed, is really given to us by the application. For example we need to cluster our course content in 5 recitations, this is the space we have. Problem “solved” 😃

In the second setting, the number of clusters is driven by the specific application, in the sense that the optimal level of the trade-off between cluster cost and K is determined by the application, for example how many colours to include in a palette vs the compression rate of the image depends from the context of the application and our needs.

Finally, the “optimal” number of clusters could be determined in a cross-validation step when clustering is used as a pre-processing tool for a supervised learning tool (for example to add a dimension “distance from the cluster centroid” when we have few labelled data and many unlabelled ones). The supervised task algorithm would then have more information to perform its separation task. Here it is the result of the supervised learning task during validation that would determine the “optimal” number of clusters.

Finally let’s have a thought on the word “unsupervised”. One common misunderstaning is that in “usupervised” tasks, as there is no labels, we don’t provide our system with any knowledge, we just “let the data speak”, decide the algorithm and let it provide with a solution. Indeed, the people who develop those unsupervised algorithms actually provide quite a bit of indirect supervision, of expert knowledge.

In clustering for example we decide about which similarity measure to use and we decide about how many clusters to give and, as we saw, ow many clusters provide. And if you’re thinking about bag of words approach of representing text, these algorithms cannot figure out which words are more semantically important than other words. So it would be up to use to do some weighting, so that the clustering results is actually acceptable.

Therefore, we should think very carefully how to do these decision choices so that our clustering is consistent with the expectation.

An implementation of K-Medoids algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L133

Lecture 15. Generative Models

15.1. Objectives

- Understand what Generative Models are and how they work
- Understand estimation and prediction phases of generative models
- Derive a relation connecting generative and discriminative models

- Derive Maximum Likelihood Estimates (MLE) for multinomial and Gaussian generative models

15.2. Generative vs Discriminative models

The model we studies up to now were **discriminative models** that learn explicit decision boundary between classes. For instance, SVM classifier, which is a discriminative model, learns its decision boundary by maximising the distance between training data points and a learned decision boundary.

At the contrary, **generative models** work by explicitly modelling the probability distribution of each of the individual classes in the training data. For instance, Gaussian generative models fit a Gaussian probability distribution to the training data in order to estimate the probability of a new data point belonging to different classes during prediction.

We will study two types of generative models, **Multinomial generative models** and **Gaussian generative models**, where for both we will ask two type of questions: (1) how do estimate the model ? How do we fit our data (the "estimation" question) and (2) how do we actually do prediction with a (fitted) model ? (the "prediction" question)

What are the advantages of generative models when compared to the discriminative ones?

For example, if your task is not just to do classification but to generate new samples of such (feature, label) pair based on the information provided in training data.

15.3. Simple Multinomial Generative model

We now think to a data-generator probabilistic model where we have different categories and hence a discrete probability distribution (a PMF, Probability Mass Function).

We name θ_w the probability for a given class w, so that $\theta_w \geq 0 \forall w$ and $\sum_w \theta_w = 1$.

Given such probabilistic model, the *generative model* $p(w|\theta)$ is nothing else than the *statistical model* to estimate the parameters θ_w given the observed data w, or more formally the statistical model described by the pair $(E, \{P_\theta\}_{\theta \in \Theta})$, where E is the sample space of the data and $\{P_\theta\}_{\theta \in \Theta})$ is the family of distributions parametrized by θ.

For example the probabilistic model may be a words generating model, given fixed vocabulary of W words and where each word would have its own probability. Then we could see a document as a serie of random (independent) extraction of these words so that a document with common words have more probabilities to be generated compared to a document made of uncommon words. As the words are independent in this model, the likelihood of the whole document to be generated is just the product of the likelihood of each world

being generated.

If we are interested in the document as a specific sequence of words, for example “IT WILL RAIN IT WILL” then it’s probability is P(“IT”) * P(“WILL”) * P(“RAIN”) * P(“IT”) * P(“WILL”) = P(“IT”)² * P(“WILL”)² * P(“RAIN”). More in general it is $P(D|\theta) = \prod_{w \in W} \theta_w^{count(w)}$. The underlying probabilistic model is then the categorical distribution, where the sample space E is then the set $1, \dots, K$ mapped to all the possible words, and $\{P_\theta\}$ is the categorical distribution parametrized by the probabilities θ. Θ, the space of the possible parameters of the probability distribution, is the set of K positive floats that sum to one.

If we are interested instead in all the possible combination of documents that use that specific number of words (in whatever order), {“IT”:2,“WILL”:2,“RAIN”:1}, we have to consider and count all the possible combinations, like “IT WILL IT WILL RAIN”, and there are $\frac{n!}{w_1!,\dots,w_i!,\dots w_k!}$ of them. The sample space E is then the set of all the possible documents, encoded as count of the different words, and $\{P_\theta\}$ is the multinomial distribution parametrized by the probabilities θ. Θ, the space of the possible parameters of the probability distribution, remains the set of K positive floats that sum to one.

Note that the categorical and multinomial distributions are nothing else than an extension of respectively the Bernoulli and binomial distributions to multiple classes (rather than just a binary 0/1 ones). They model the probability of respectively one or multiple, independent, categorical trials, i.e. the outcome for the categorical distribution is the single word, while those for the multinomial distribution is the whole document (encoded as world count).

For completion, there are $\frac{n!}{w_1!,\dots,w_i!,\dots w_k!}$ possible combinations of sequences of a document with $w_1, \dots, w_i, \dots w_k$ worlds count (with $\sum_{i=1}^k w_i = n$), so the PMF of the multinomial is
$p(w_1, .., w_i, \dots, w_k; \theta_1, \dots, \theta_i, \dots, \theta_k) = \frac{n!}{w_1!,\dots,w_i!,\dots w_k!} \prod_{i=1}^k \theta_i^{w_i}$.

15.4. Likelihood Function

Even if we will use the term “multinomial generative model”, in this lecture we will consider the first approach of encoding a document and its relative probabilistic model (the categorical distribution)

Note indeed that in some fields, such as machine learning and natural language processing, the categorical and multinomial distributions are conflated, and it is common to speak of a “multinomial distribution” when a “categorical distribution” would be more precise.

So, for a particular document D it’s probability to be generated by a sequence of samples from a categorical distribution with parameter θ is
$P(D|\theta) = \prod_{w \in W} \theta_w^{count(w)}$.

Given an observed document and compelling probabilistic models (with different thetas) we can then determine wich is the one with the highest probability of

having generated the observed document.

Let's consider for example a vocabulary of just two words, "cat" and "dog" and an observed Document "cat cat dog". Let's also assume that we have just two compelling models to choose from. The first (probabilistic) model has $\theta_{\text{cat}} = O.3$ and $\theta_{\text{dog}} = O.7$. The second model has $\theta_{\text{cat}} = O.9$ and $\theta_{\text{dog}} = O.1$ It is intuitive that it is more probable that is the second model that generated the document, but let's compute it. The probability of the document being generated by the first model is $0.3^2 * 0.7 = 0.0189$. The probability that D has been generated instead by the second model is $0.9^2 * 0.1 = 0.081$, that is higher than those for the first model.

15.5. Maximum Likelihood Estimate

The joint probability of a set of given outcomes when we want to stress it as a function of the parameters of the probabilistic model (thus treating the random variables as fixed at the observed values) is known as the likelihood.

While often used synonymously in common speech, the terms "likelihood" and "probability" have distinct meanings in statistics. *Probability* is a property of the sample, specifically how probable it is to obtain a particular sample for a given value of the parameters of the distribution; *likelihood* is a property of the parameter values.

We want now to find which are the parameters that maximise the likelihood.

For the multinomial model it is $\text{argmax}_\theta\{L(D|\theta_0) = \prod_{w \in W} \theta_w^{\text{count}(w)}\}$.

It turns out that maximising its log (known as the log-likelihood) is computationally simpler while equivalent (in terms of the argmax, not in terms of its value) because the log function is a monotonically increasing function: wherever the likelihood is on its maximum, its log it is on its maximum as well.

We hence compute the first order conditions in terms of theta for the log-likelihood to find the theta that maximise it:

$\text{argmax}_\theta\{lL(D|\theta_0) = \sum_w \text{count}(w) log(\theta_w)\}$

Max likelihood estimate for the cat/dog example

Let's first consider the specific case of just two categories to later generalise (and let's calling the outcomes just 0/1 instead of cat/dog).

In such setting we have really just one parameter, θ_0, (the probability of a 0) as we can write θ_1 as $1 - \theta_0$ The log_Likelihood of a given document is then:

$lL(D|\theta_0) = \text{count}(0) log(\theta_0) + \text{count}(1) log(1 - \theta_0)$

Taking the derivative with respect to θ_0 and setting it equal to zero we obtain:

$$\frac{\partial lL}{\partial \theta_0} = \frac{\text{count}(0)}{\theta_0} - \frac{\text{count}(1)}{1-\theta_0}$$

$$\frac{\partial lL}{\partial \theta_0} = 0 \rightarrow \tilde{\theta}_0 = \frac{\text{count}(0)}{\text{count}(0)+\text{count}(1)}$$

Note the symbol of the tilde to indicate an estimated parameter. This is our "best guess" parameter given the observed data.

15.6. MLE for Multinomial Distribution

Let's now consider the full multinomial case. So given a document (or, as the words are assumed to be independent, a whole set of documents... just concatenated one to the other) and a vocabulary W we want to find $\theta_w \forall w \in W$ as to maximise {L(D|\theta_0) = \prod_{w \in W} \theta_w^{\text{count} (w)}}$.

The problem is however that the thetas are not actually free, but we have the constrain that the thetas have to sum to one (we have actually also those that the thetas need to be positive, but for the nature of this maximisation problem we can ignore this constraint).

The method of Lagangian multipliers for constrained optimisation

We have hence a constrained optimisation problem that we can solve with the method of the **Lagrange multipliers**, a method to find the stationary values (min or max) of a function subject to *equality* constraints.

Let's consider a case of an objective function of two variables $z = f(x, y)$ subject to a single constraint $g(x, y) = c$, where c is a constant (both the objective function and the constraint doesn't need to be necessarily linear).

We can then write the so-called *Lagrangian function* as $\mathscr{L} = f(x, y) + \lambda[c - g(x, y)]$, i.e. we "augment" the original function we want to find the extremes with a term made by a new variable (the Lagrangian multiplier) that multiplies the relative constraint (if we have multiple contraints, we would have multiple additional terms and Lagrangian multipliers).

All we need to do now is to find the stationary values of $\mathscr{L}$, regarded as a function of the original variables x and y but also of the new variable(s) λ we introduced.

The First Order necessary Condition (FOC) are:

$$\begin{aligned}
\mathscr{L}_x &= f_x - \lambda g_x \quad &= 0 \\
\mathscr{L}_y &= f_y - \lambda g_y \quad &= 0 \\
\mathscr{L}_\lambda &= c - g(x, y) &= 0
\end{aligned}$$

Solving this system of 3 equations in 3 unknown will equivalently find the stationary point of the unconstrained function $\mathscr{L}$ and original function f, as the

third equation in the FOC guarantee that indeed the constrain is respected and, at the stationary point, the two functions have the same value.

For a numerical example, let's the function to optimize be $z = xy$ and be it subjected tothe constraint $x + y = 6$. We can write the Lagrangian as $\mathscr{L} = xy + \lambda[6 - x - y]$ whose FOC are:

$$\begin{aligned}\mathscr{L}_x &= y - \lambda &&= 0\\ \mathscr{L}_y &= x - \lambda &&= 0\\ \mathscr{L}_\lambda &= 6 - x - y &&= 0\end{aligned}$$

Solving for (x, y, λ) we find the optimal values $x^* = 3$, $y^* = 3$, $\lambda^* = 3$ and $z^* = 9$.

The constrained multinomial log-likelihood

As the sum of thetas must be one, the Lagrangian of the log-likelihood is:

$$\mathscr{L} = \log P(D|\theta) + \lambda\left(\sum_{w \in W} \theta_w - 1\right) = \sum_{w \in W} n_w \log \theta_w + \lambda\left(\sum_{w \in W} \theta_w - 1\right)$$

where n_w is the count of word w in the document.

The FOC of the lagrangian are then:

$$\begin{aligned}\mathscr{L}_w &= \frac{n_w}{\theta_w} - \lambda &&= 0\\ \mathscr{L}_\lambda &= \sum_{w \in W} \theta_w - 1 &&= 0\end{aligned}$$

Where the first equation is actually a gradient with respect to all the w.

Solving the above system of equation we obtain $\hat{\theta}_w = \frac{n_w}{\sum_{w \in W} n_w} \forall w \in W$.

These set of θ_w parameters are the **maximum likelihood estimates**, the values of θ that maximize the likelihood function for the observed data and this multinomial generative model.

15.7. Prediction

Let's now see how can we actually use generative Multinomial Models to make predictions, for example deciding if a document is in Spanish of Portuguese language.

For that we have first given a large set of documents in both languages, together with their label (the language).

From there we can estimate the PMF of the words using the MLE estimator (i.e., counting the relative proportions) for both Spanish and Italian (so our vocabulary would have both Spanish and Italian terms).

We are now given a new document without the language label. Calling the two languages as “+” and “-”, we compute the likelihood of the document under the first hypothesis ($P(D|\theta^+)$) as the joint probability using the first PMF , and the likelihood under the second hypothesis ($P(D|\theta^-)$) using the second PMF, and we decide how to classify the document according to which one is bigger.

The probability of a data item to belong to a given class, conditional to the probabilities of its feature vector is known also as class-conditional probability.

We can equivalently say that we category the document as “+” if $log(\frac{P(D|\theta^+)}{P(D|\theta^-)}) > 0$.

But $\log \frac{P(D|\theta^+)}{P(D|\theta^-)} = \log P(D|\theta^+) - \log P(D|\theta^-)$

$= \log \prod_{w \in W} (\theta_w^+)^{\text{count}(w)} - \log \prod_{w \in W} (\theta_w^-)^{\text{count}(w)}$

$= \sum_{w \in W} \text{count}(w) \log \theta_w^+ - \sum_{w \in W} \text{count}(w) \log \theta_w^-$

$= \sum_{w \in W} \text{count}(w) \log \frac{\theta_w^+}{\theta_w^-}$

$= \sum_{w \in W} \text{count}(w) \hat{\theta}_w$

where $\hat{\theta}_w$ is just a symbol for $\log \frac{\theta_w^+}{\theta_w^-}$.

The last expression shows that this classifier, derived looking through a generative view on classification, should actually remind us a linear classifier that goes through origin with respect to this parameter $\hat{\theta}_w$.

15.8. Prior, Posterior and Likelihood

Let’s now try to compute $P(\theta|D)$, i.e. the probability of the language of the document (that is the probability of the generating PMF being those of the Spanish or Portuguese language) given the observed document.

To compute it, we will apply the **Bayesian rule** that states:

$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$

and the **total probability law** that states:

$P(X = x) = \sum_y P(Y = y) * P(X = x|Y = y)$

$P(lang = spanish|D) = \frac{P(D|lang=Spanish) * P(lang=Spanish)}{P(D)}$

Where
$P(D) = P(D|lang = Spanish) * P(lang = Spanish) + P(D|lang = Portuguese) * P$

We can think for example to a bilingual colleague giving us a document in either Spanish or Portuguese. $P(lang = Spanish)$ is the **prior** probability that the document is Spanish (because maybe we know that this guy works more with Spanish documents than Portuguese ones). Then $P(D|lang = Spanish)$ and $P(D|lang = Portuguese)$ are the two conditional probabilities that depends on the world frequencies of the two languages respectively and $P(D)$ is the probability that our colleague gave us exactly that document.
$P((lang = spanish|D)$ is known as the **posterior** probability, as it is the one we have once we observe document D.

Going back to our plus/minus class notation instead of the languages, we can compute the log of the ratio between the posteriors for the two classes as:

$$log(\frac{P(y=+|D)}{P(y=-|D)}) = log(\frac{P(D|y=+)*P(y=+)}{P(D|y=-)*P(y=-)}) = log(\frac{P(D|y=+)}{P(D|y=-)}) + log(\frac{P(y=+)}{P(y=-)})$$

Using the expression from the previous segment for the first term and $\hat{\theta}_0$ as a symbol representing the second term, we can rewrite the equation as:

$$log(\frac{P(y=+|D)}{P(y=-|D)}) = \sum_{w \in W} \text{count}(w)\hat{\theta}_w + \hat{\theta}_0.$$

So now what we actually see here that we translated it again to a linear classifier. But in contrast to the previous case, when we had a linear classifier that went through origin, now we have a linear classifier with an offset, and the offset itself would be actually guided by our prior, which will drive the location of the separator.

So what we've seen here that in this model we can very easily incorporate our prior knowledge about the likelihood of certain classes. And at the end, what we got, that even though we're talking about generative models and we're using a different mechanism on some ways of estimating the parameters of this multinomial, at the end, we actually are getting the same linear separators that we see in our discriminative modelling.

15.9. Gaussian Generative models

We will now switch, in place to a categorical distribution, to a generative (probabilistic) model based on the multidimensional Normal (or "Gaussian") distribution.

While the categorical distribution was opportune for modelling discrete random variables, like classes, the Normal is the analogous counterpart for continuous random variables.

The multidimensional Normal has Probability Density Function (PDF):

$$p_{\mathbf{X}}(\mathbf{x}; \mu, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu)}, \quad \mathbf{x} \in \mathbb{R}^d$$

where $\mathbf{x} \in \mathbb{R}^d$, $\mu \in \mathbb{R}^d$ is the vector of the means in the d dimensions and $\mathbf{\Sigma} \in \mathbb{R}^{d \times d}$ is the covariance matrix (with $\det(\Sigma)$ being its determinant).

So the PDF has a single spike in correpondance of $x = \mu$ and then gradually declines in a typical “bell shape”. The “speed” of the decline is regulated by the sigma parameter: more sigma is high, more the spike is low and the tails are “fat”; making the whole curve “flatted”.

When all the components (dimensions) are uncorrelated (i.e. the covariant matrix Σ is diagonal) and have the same standard deviation σ, the Normal PDF reduces to:

$$p_{\mathbf{X}}(\mathbf{x}; \mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2} * ||x-\mu||^2}$$

We should also specify that a random vector $\mathbf{X} = (X^{(1)}, \ldots, X^{(d)})^T$ is defined as a **Gaussian vector**, or “multivariate Gaussian” or “normal variable”, if any linear combination of its components is a (univariate) Gaussian variable or a constant (a “Gaussian" variable with zero variance), i.e., if $\alpha^T \mathbf{X}$ is (univariate) Gaussian or constant for any constant non-zero vector $\alpha \in \mathbb{R}^d$.

It is important to note that in generative models we are introducing a specific functional form for the probabilistic model (here the gaussian), and our freedom relates only on the parameters of the model, not on the form itself. Hence we first need to consider if our data, our specific case, is appropriate to be modelled by such functional form or not.

15.10. MLE for Gaussian Distribution

As for the multinomial case, if we have a training set $S_n = \{x^{(t)} | t = 1...n\}$ we can compute the -log-likelihood of this set and find the (μ, σ^2) that maximise it:

The likelihood is $p(\mu, \sigma^2; S_n) = \prod_{t=1}^n p(\mu, \sigma^2; x^{(t)}) = \prod_{t=1}^n \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2} * ||x-\mu||^2}$

The log-likelihood is then:

$$logp(\mu, \sigma^2; S_n) = log(\prod_{t=1}^n \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2} * ||x^{(t)}-\mu||^2})$$

$$= -\frac{nd}{2} log(2\pi\sigma^2) + \frac{1}{2\sigma^2} ||x^{(t)} - \mu||^2$$

From this expression we can take the derivatives of μ and σ and equal them to zero to find the MLE estimators $\hat{\mu} = \frac{1}{n} \sum_{t=1}^n x^{(t)}$ and $\hat{\sigma}^2 = \frac{\sum_{t=1}^n ||x^{(t)}-\mu||^2}{nd}$.

Lecture 16. Mixture Models; EM algorithm

16.1. Mixture Models and the Expectation Maximization (EM) Algorithm

Objectives:

- Review Maximum Likelihood Estimation (MLE) of mean and variance in Gaussian statistical model
- Define Mixture Models
- Understand and derive ML estimates of mean and variance of Gaussians in an Observed Gaussian Mixture Model
- Understand Expectation Maximization (EM) algorithm to estimate mean and variance of Gaussians in an Unobserved Gaussian Mixture Model

16.2. Recap of Maximum Likelihood Estimation for Multinomial and Gaussian Models

So far, in clustering we have assumed that the data has no probabilistic generative model attached to it, and we have used various iterative algorithms based on similarity measures to come up with a way to group similar data points into clusters. In this lecture, we will assume an underlying probabilistic generative model that will lead us to a “natural” clustering algorithm called the **EM algorithm** .

While a “hard" clustering algorithm like k-means or k-medoids can only provide a cluster ID for each data point, the EM algorithm, along with the generative model driving its equations, can provide the posterior probability (“soft" assignments) that every data point belongs to any cluster.

Today, we will talk about mixture model and the EM algorithm. The rest of the segment is just a recap of the two probabilistic models we saw, the multinomial (actually the categorical) and the multidimansional Gaussian.

The corresponding statistical model is, given observed data and the assumption that the data has been generated by the given probabilistic model, which is the most likely parameter of the distribution, like the individual category probabilities θ_w for the categorical or (μ, σ^2) for the Gaussian ?

This is known in statistics as parametric estimation, and can be solved by interpreting the joint probability function of a set of observed data in terms of a function of the parameter of the model given the observed data (and in doing this we name the joint probability function “likelihood”) and then find the parameter(s) of the model that maximise it, using first order conditions, eventually employing Lagrange Multipliers if the maximisation involves respecting some contraints, like setting the sum of all the categorical probabilities equal to one.

Please also consider that the sum of independent normal random variables remains normal:

$$X \sim N(\mu, \sigma^2) \rightarrow aX + b \sim N(a\mu + b, a^2\sigma^2)$$

16.3. Introduction to Mixture Models

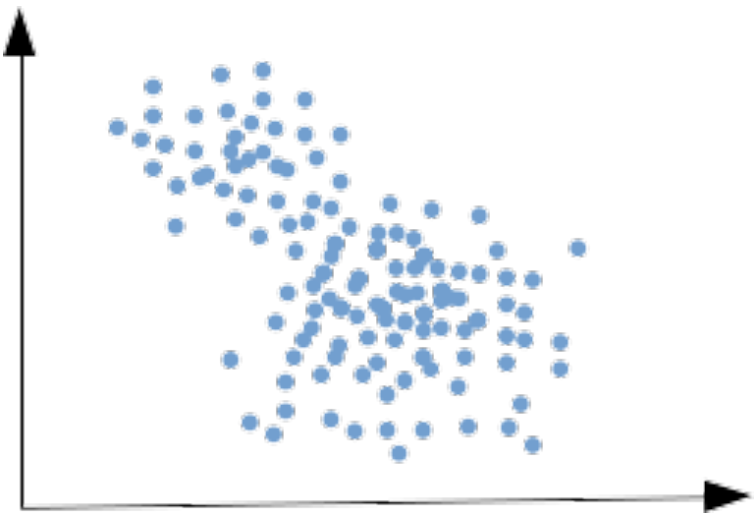
Let's know consider a generative (probabilistic) model consisting of two subsequent steps. We first draw a class using a K-categorical distribution with probabilities $p_1, \ldots, p_j, \ldots p_K$.

We then draw the final data from a Normal distribution with parameter ($x^{(j)}$ and $\sigma^{2^{(j)}}$), i.e. specific of the class sampled in the first step.

This generative model goes under the name of **mixture model** and mixture models are actually very useful for describing many real-life scenarios.

The full set of parameters of this mixture model is then collectively represented by $\theta = \{p_1, \ldots, p_j, \ldots p_K; \mu^{(1)}, \ldots, \mu^{(j)}, \ldots, \mu^{(K)}, \sigma^{2^{(1)}}, \ldots, \sigma^{2^{(j)}}, \ldots, \sigma^{2^{(K)}}\}$ where the non-negative p_j are called **mixture weights** (and sum to 1) and the individual Gaussians with parameters $\mu^{(j)}$ and $\sigma^{2^{(j)}}$ are the **mixture components** (so this is more specifically a **Gaussian mixture model (GMM)**, not to be confonded with the "Generalized Method of Moments"). K is the size of the mixture model.

The output of sampling a set of points from such model, using 2 components (two classes of the categorical distribution) and a two-dimensional Gaussian, would look something like this:



We can intuitively distinguish the two clusters and see that one is bigger than the other, both in terms of points (i.e. its p_j is larger) and in term of spread (its $\sigma^{2^{(j)}}$ is bigger).

But where is the border between the two points? While the K-Mean or K-Medoids algorithm would have provided a classifier, returning one cluster id for each point (**hard clustering model**), a clustering based on a mixture generative model (to be implemented using the EM algorithm we will discuss in the next segment) would instead return for each point the *probability* to belong to each cluster (**soft clustering model**). For example, the points in the top-left corner of the picture will have probability to belong to the first cluster almost one and those in the bottom-right corner probability to belong to the second cluster (i.e. being generated by a Normal distribution with parameters (μ_2, σ_2^2)) almost 1,

but those in the center will have much more balanced probabilities.

So here, we have a much more refined way to talk about our distribution, because every point kind of belongs to different components, just with different likelihoods.

If we know all of the parameters of the K Gaussians and the mixture weights, the probability density function $p(x|\theta)$ can be computed using the law of total probability as $p(\mathbf{x}|\theta) = \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}; \mu^{(j)}, \sigma_j^2)$.

Using the Bayes rule we can also find the posterior probability that $\mathbf{x}$ belongs to a Gaussian component j.

Likelihood of Gaussian Mixture Model

Given the GMM we can find the PDF associated with each point x using the total probability theorem: $p(\mathbf{x}; \theta) = \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}, \mu^{(j)}, \sigma_j^2)$

The likelihood associated to a set S_n of observed, independent, data is then $p(\theta, S_n) = \prod_{i=1}^{n} \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}^{(i)}, \mu^{(j)}, \sigma_j^2)$

Now, given this expression, we want to take the derivatives of it with respect to my parameters, make them equal to 0, and find the parameters. It turns out however that's actually a pretty complex task.

We will first start to reason in a setting where we know the category associated to each point, to further generalise in a context where we don't know the class from which it belongs, with the EM algorithm.

Note also that while we will consider different means of the Gaussian across its dimensions (i.e. $\mu^{(j)}$ is a vector) we will assume the same variance across them (i.e. σ_j^2 is a scalar).

16.4. Mixture Model - Observed Case

Let's hence first assume that we know the cluster j to which each point i belong to and we need to estimate the parameters of the likelihood using MLE.

For that let's define as **indicator function** a function that, for a given pair (i, j), returns 1 if the j cluster is the one to with point i belongs to, 0 otherwise.

$$\mathbf{1}(j; i) := \begin{cases} 1 & \text{if } x^{(i)} \in \text{cluster}_j, \\ 0 & \text{if otherwise.} \end{cases}$$

The log-likelihood can then be rewritten using such indicator function and inverting the summation as:

$$log p(\theta, S_n) = \sum_{j=1}^{K} \sum_{i=1}^{n} \mathbf{1}(j; i) log[p_j \mathcal{N}(\mathbf{x}^{(i)}, \mu^{(j)}, \sigma_j^2)]$$

Note that in general $log(sum(x))$ is not the same as $sum(log(x))$, but the presence of the indication function guarantees that in reality there is no summation over the cluster when we have a specific data example.

And what this expression actually demonstrates is that, whenever we are trying to find all the parameters that optimize for each mixture component, we can actually do this computation for each cluster independently. We can independently find the best parameters that fit the cluster, because the fact that these points are observed, we actually know where the point is belonging to. So we can separately find, here the mu, sigma and the p.

By maximising the likelihood, we would find the following estimators:

- $\hat{n}_j$, the number of points of each cluster: $\hat{n}_j = sum_{i=1}^{n} \mathbf{1}(j;i)$ (this actually just derives from our assumption to know the indicator function output)
- $\hat{p}_j = \frac{\hat{n}_j}{n}$
- $\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{i=1}^{n} \mathbf{1}(j;i) * x^{(i)}$
- $\hat{\sigma}_j^2 = \frac{1}{\hat{n}_j d} \sum_{i=1}^{n} \mathbf{1}(j;i) * ||x^{(t)} - \mu^{(j)}||^2$

These equations show how, given the observed case, when we know to which component each point belong, we can estimate all the parameters that we need to define our mixture of Gaussian.

16.5. Mixture Model - Unobserved Case: EM Algorithm

We now consider the unobservated case, where we don't know the mapping between data x_i and cluster j.

The notation of the indicator function implies a hard counting: the point belongs to one cluster with certainty or it doesn't belong. But one of the powers of mixture model is that we can see each point to doesn't solely belong to one single cluster, we know that it can be generated from different ones with different probabilities. So now, instead of talking about hard counts - belong or doesn't - we'll actually talk about **soft counts**: how much does this point belongs to this particular cluster?

What we want is $p(j|x^{(i)})$: the probability that, having observed point i, this would have been generated by cluster j.

If we would know all the parameters we could compute it using the Bayes rule:

$$p(j|x^{(i)}) = \frac{p(j,x^{(i)})}{p(x^{(i)})} = \frac{p(j)*p(x^{(i)}|j)}{\sum_{k=1}^{K} p(k)*p(x^{(i)}|k)}$$

Indeed $p(j)$ would be just our p_j and $p(x^{(i)}|j)$ would be $N(\mu_j, \sigma_j^2)$.

But unfortunately there is no closed solution for maximising the likelihood $p(\theta, S_n)$ involving the full set of parameters, even with a very large S_n observed,

as the parameters are very interconnected.

We need hence to break these dependence with a solving procedure in two parts: given my best estimated parameters, I first compute $p(j|x^{(i)})$. Then, treating the estimated $p(j|x^{(i)})$ as given, I maximise the likelihood $p(\theta, S_n)$ to retrieve the parameters. Solving for the parameters I would then find something very similar to the results in the previous segment:

- $\hat{n}_j$, the number of points of each cluster: $\hat{n}_j = sum_{i=1}^{n} p(j|i)$ (this is actually a weighted sum of the points, where the weights are the relative probability of the points to belong to cluster j)
- $\hat{p}_j = \frac{\hat{n}_j}{n}$
- $\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{i=1}^{n} p(j|i) * x^{(i)}$
- $\hat{\sigma}_j^2 = \frac{1}{\hat{n}_j d} \sum_{i=1}^{n} p(j|i) * ||x^{(t)} - \mu^{(j)}||^2$

At this point I can re-compute the $p(j|x^{(i)})$ and so on until my results don't change any more or change very little.

Finding the posterior $p(j|x^{(i)})$ given the parameters and deriving the expected likelihood under this $p(j|x^{(i)})$ is called the **E step** (for "expectations"), while then finding the parameters by maximising the likelihood given the estimated $p(j|x)$ is called the **M step** (for "maximisation"). Note that the EM algorithm is not specific to the GMM, but it can be used any time we need to estimate the parameters in a statistical models where the model depends on unobserved latent variables.

Of course we are still left with the problem of how to initiate all the parameters for the first E step. The EM algorithm indeed is very sensitive to initial conditions as it guarantee to find only a *local* maximisation of the $p(\theta, S_n)$, not a global one.

We could either do a random initialization of the parameter set θ or we could employ k-means to find the initial cluster centers of the K clusters and use the global variance of the dataset as the initial variance of all the K clusters. In the latter case, the mixture weights can be initialized to the proportion of data points in the clusters as found by the k-means algorithm.

An implementation of E-M algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L222

Homework 5

Project 4: Collaborative Filtering via Gaussian Mixtures

P4.3. Expectation–maximization algorithm

Data Generation Models

[MITx 6.86x Notes Index]