Contents

Chapter 1

Introduction

1.1 Quick refresher: Supervised learning

In the previous semester, we had the opportunity to study the concepts of supervised learning, the mathematical structures underneath it and analyze various models like the Perceptron, Adaline, SVM and so on. The basic idea was to use the N pre-labeled data points $(x^{(i)}, y^{(i)})_{1 \leq i \leq N}$ and build a hyperplane that would separate (in the "one versus the rest" case) one class of data points from the other class(es). The points $x^{(i)} \in \mathbb{R}^{n+1}$ were called *data points*, each having n characteristic features (where the first component was chosen to be 1 by convention), whereas $y^{(i)}$ were called *labels*. The labels can have, in the case of a binary classification problem, the values -1 and 1. In the multiclass regime, the labels can take a value from a larger range of values. Afterwards, we would insert other data points $(x^{(i)})_{i>N}$ and expect our algorithm to classify them correctly, i.e. label them with the correct value $(y^{(i)})_{i>N}$.
So, we formulated the binary classification problems: given pre-labeled data, we want to find a hyperplane (which is characterized by a vector normal to the plane, which we usually called w) which separates accurately our pre-labeled data, and also successfully separates new data. We took a step further by introducing a certain class of functions, which we called *activation functions* (Adaline), which would help us improve our algorithm, and afterwards allowing us to use the tools of optimization theory, defining loss functions, minimizing them, and so on (like we did for the SVM).

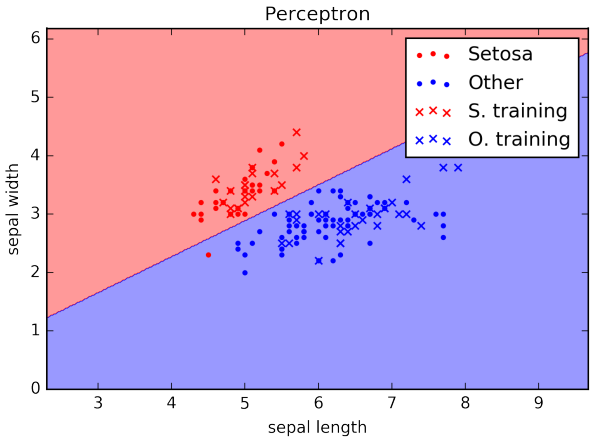


Figure 1.1: The Perceptron model in action

1.2 The basic idea of Unsupervised learning

Unsupervised learning is a branch of machine learning that learns from test data that has not been labeled, classified or categorized. Instead of responding to feedback, unsupervised learning identifies commonalities in the data and reacts based on the presence or absence of such commonalities in each new piece of data.
The data given to unsupervised algorithm are not labelled, which means only the input variables, i.e. the data points $x^{(i)}$ are given with no corresponding output variables. So the machine has to learn on its own to "separate data", without any training. This is the first and most important difference in comparison to supervised learning.
The first idea that one could come up to would be to **visualize** the data, and the person would be in charge for finding structure/patterns in the data points. This is, however, a problem if the number of data points gets huge, and not to mention the problem that the feature space cold have a dimension larger than 3. But suppose that the number of dimensions is the only problem. Another method that could work is called **dimensional reduction**. Using this method, we focus only on the relevant dimensions. However, we would then need a criterion for defining "relevant" and "irrelevant" dimensions.
So the problem must be tackled with some other methods. Let us therefore look at a simple example of non-labeled data points:

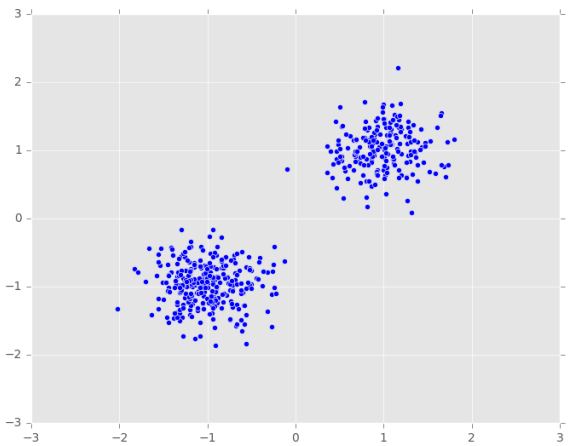


Figure 1.2: Two *clusters* of unlabeled data points

We clearly see that there are (at least) two "bunches" of data, so-called *clusters*. But how can we quantitatively conclude that there are two clusters? On one hand, for one cluster, there is a certain domain of the feature space characteristic for it. On the other hand, we can measure the **distance** between the points; the distance between the points that belong to the same cluster is smaller than the distance between two points that are from different clusters. This will be a guiding principle for building up algorithms.
Some of the most popular algorithms for unsupervised learning include **clustering algorithms**, among which are the K-means clustering algorithm, hierarchical clustering, DBSCAN, mixture models and so on. Other algorithms work on the principle of **anomaly detection** (like the Local Outlier Factor algorithm). Of course, there are far more algorithms than we have mentioned with other underlying principles, which we will briefly discuss later.
The algorithms for these methods can be found in the Python module scikit-learn. scikit-learn is a Python module for machine learning built on top of SciPy: `https://github.com/scikit-learn`

Chapter 2

Cluster analysis

Cluster analysis groups data objects based only on information found in the data that describes the objects and their relationships. The goal is that the objects within a group be similar (or related) to one another and different from (or unrelated to) the objects in other groups. The greater the similarity (or homogeneity) within a group and the greater the difference between groups, the better or more distinct the clustering.
An entire collection of clusters is commonly referred to as a clustering, and in this section, we distinguish various types of clusterings: hierarchical (nested) versus partitional (unnested), exclusive versus overlapping versus fuzzy, and complete versus partial.

A **partitional** clustering is simply a division of the set of data objects into non-overlapping subsets (clusters) such that each data object is in exactly one subset. If we permit clusters to have subclusters, then we obtain a **hierarchical** clustering, which is a set of nested clusters that are organized as a tree. Each node (cluster) in the tree (except for the leaf nodes) is the union of its children (subclusters), and the root of the tree is the cluster containing all the objects. **Exclusive** clusters assign each object to a single cluster. There are many situations in which a point could reasonably be placed in more than one cluster, and these situations are better addressed by **non-exclusive** clustering. In the most general sense, an overlapping or non-exclusive clustering is used to reflect the fact that an object can simultaneously belong to more than one group (class). In a **fuzzy** clustering, every object belongs to every cluster with a membership weight that is between 0 (absolutely doesn't belong) and 1 (absolutely belongs). With such a clustering, we address situations where data points could belong to more clusters. Usually, the fuzzy clustering is combined with partitional clustering, by assigning data points to the clusters with the highest probability.

Now that we have defined the basic terms, let us investigate some algorithms.

2.1 K-means clustering

In this section, we will mainly follow Bishop (2006)[1].
Suppose we have a data set of N data points $(x^{(i)})_{1\leq i\leq N}$ which live in the d-dimensional feature space. Our goal is to partition the data set into some number K of clusters, where we shall suppose for the moment that the value of K is given. We can formalize this notion by first introducing a set of n-dimensional vectors $\boldsymbol{\mu}_k$, where $k = 1, ..., K$, in which $\boldsymbol{\mu}_k$ is a *prototype* associated with the k^{th} cluster. As we shall see shortly, we can think of the $\boldsymbol{\mu}_k$ as representing the centres of the clusters. Our goal is then to find an **assignment of data points to clusters**, as well as a **set of vectors $\boldsymbol{\mu}_k$**, such that the sum of the squares of the distances of each data point to its closest vector $\boldsymbol{\mu}_k$, is a minimum.
It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point $x^{(i)}$, we introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, ..., K$ describing which of the K clusters the data point $x^{(i)}$ is assigned to, so that if data point $x^{(i)}$ is assigned to cluster k, then $r_{nk} = 1$, and $r_{nj} = 0$ for $j = k$. This is known as the 1-of-K coding scheme. We can then define a loss function, sometimes called a distortion measure, given by:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2 \tag{2.1}$$

which represents the sum of the squares of the distances of each data point to its assigned vector $\boldsymbol{\mu}_k$. Our goal is to find values for the r_{nk} and the $\{\boldsymbol{\mu}_k\}$ so as to minimize J. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{nk} and the $\boldsymbol{\mu}_k$.
First we choose some initial values for the $\boldsymbol{\mu}_k$. Then in the first phase we minimize J with respect to the r_{nk}, keeping the $\boldsymbol{\mu}_k$ fixed. In the second phase we minimize J with respect to the $\boldsymbol{\mu}_k$, keeping r_{nk} fixed. This two-stage optimization is then repeated until convergence. We shall see that these two stages of updating r_{nk} and updating $\boldsymbol{\mu}_k$ correspond respectively to the E (expectation) and M (maximization) steps of the EM algorithm[2], and to

[1] Christopher M. Bishop - *Pattern Recognition and Machine Learning*, Springer Verlag

[2] The EM algorithm is a much broader and more general algorithm, but we will not deal with this algorithm in detail

emphasize this we shall use the terms E step and M step in the context of the K-means algorithm.
Consider first the determination of the r_{nk}. Because J in 2.1 is a linear function of r_{nk}, this optimization can be performed easily to give a closed form solution. The terms involving different n are independent and so we can optimize for each n separately by choosing r_{nk} to be 1 for whichever value of k gives the minimum value of $\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$. In other words, we simply assign the n^{th} data point to the closest cluster centre. More formally, this can be expressed as

$$r_{nk} = \begin{cases} 1 & \text{if } k = \text{argmin}_j \left\|\mathbf{x}_n - \boldsymbol{\mu}_j\right\|^2 \\ 0 & \text{otherwise.} \end{cases}$$

Now consider the optimization of the $\boldsymbol{\mu}_k$ with the r_{nk} held fixed. The loss function j is a quadratic function of $\boldsymbol{\mu}_k$ and it can be minimized by setting its derivative with respect to $\boldsymbol{\mu}_k$ to zero. This will give us:

$$2\sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0$$

which we can easily solve for $\boldsymbol{\mu}_k$:

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\mathbf{x}_n}{\sum_n r_{nk}} \tag{2.2}$$

The denominator in this expression is equal to the number of points assigned to cluster k, and so this result has a simple interpretation, namely set $\boldsymbol{\mu}_k$ equal to the mean of all of the data points $\mathbf{x}_n$ assigned to the cluster k. For this reason, the procedure is called K-means algorithm.

The two phases of dd-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J, convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J. The convergence properties of the K-means algorithm were studied by MacQueen (1967)[3]. Let us now summarize how the algorithm looks like and also prove the convergence of the algorithm.

[3]J. MacQueen - *Some methods for classification and analysis of multivariate observations,* `https://projecteuclid.org/euclid.bsmsp/1200512992`

K-means algorithm (pseudocode)

```
1    for k=1 to K do
2    μ_k ← some random location   randomly initialize mean for kth cluster
3    end for
4    repeat
5     for n = 1 to N do
6     r_nk ← argmin_k ||x_n − μ_k||^2   E-step: assign n-th data point to closest center
7     end for
8     for k = 1 to K do
9       μ_k ← MEAN_k(x_n, r_nk)   M-step: re-estimate mean of cluster k with 2.2
10    end for
11   until converged
12   return r   return cluster assignments
```

Theorem (K-means Convergence Theorem): For any dataset D and any number of clusters K, the K-means algorithm converges in a finite number of iterations, where convergence is measured by J ceasing the change.

We follow the proof given by Hal Daumé III in his book *A Course in Machine Learning.*

Proof: The proof works as follows. There are only two points in which the K-means algorithm changes the values of $\boldsymbol{\mu}_k$ or r_{nk}: lines 6 and 9. We will show that both of these operations can never increase the value of J. Assuming this is true, the rest of the argument is as follows. After the first pass through the data, there are only finitely many possible assignments to r_{nk} and $\boldsymbol{\mu}_k$, because r_{nk} is discrete and because $\boldsymbol{\mu}_k$ can only take on a finite number of values: means of some subset of the data. Furthermore, J is lower-bounded by zero (since J is just a sum of Euclidean (L_2) distances). Together, this means that J cannot decrease more than a finite number of times. Thus, it must stop decreasing at some point, and at that point the algorithm has converged. It remains to show that lines 6 and 9 decrease J. For line 6, when looking at example n, suppose that the value of r_{na} was 1, and now r_{nb} is 1, and r_{na} is zero. It must be the case that $\|\mathbf{x}_n - \boldsymbol{\mu}_b\| \leq \|\mathbf{x}_n - \boldsymbol{\mu}_a\|$. Thus, changing from a to b can only decrease J. For line 9, consider the second form of J. Line 9 computes $\boldsymbol{\mu}_b$ as the mean of the data points for which $r_{nb} = 1$, which are precisely the points that minimize the squared distances. Thus, this update to $\boldsymbol{\mu}_k$ can only decrease J. □

There are several aspects of K-means that are unfortunate. First, the convergence is only to a local optimum of J. In practice, this means that you

should usually run it 10 times with different initializations and pick the one with minimal resulting J. Second, one can show that there are input datasets and initializations on which it might take an exponential amount of time to converge. Fortunately, these cases almost never happen in practice, and in fact it has recently been shown that (roughly) if you limit the floating point precision of your machine, K-means will converge in polynomial time (though still only to a local optimum), using techniques of smoothed analysis.[4]

2.1.1 Variations of the K-means algorithm

A direct implementation of the K-means algorithm as discussed here can be relatively slow, because in each E step it is necessary to compute the Euclidean distance between the prototype vector and every data point. Various schemes have been proposed for speeding up the K-means algorithm, some of which are based on precomputing a data structure such as a tree such that nearby points are in the same subtree (Ramasubramanian and Paliwal, 1990; Moore, 2000). Other approaches make use of the triangle inequality for distances, thereby avoiding unnecessary distance calculations (Hodgson, 1998; Elkan, 2003).
The biggest practical issue in K-means is initialization. If the cluster means are initialized poorly, you often get convergence to uninteresting solutions. It can also happen that we have two centroids concentrated in one cluster. A useful heuristic is the furthest-first heuristic. This gives a way to perform a semi-random initialization that attempts to pick initial means as far from each other as possible. The heuristic is sketched below:

- Pick a random example m and set $\boldsymbol{\mu}_1 = x^{(m)}$.
- For $k = 2, .., K$: Find the example m that is as far as possible from all previously selected means; namely: $m = \text{argmax}_m \text{min}_{k' < k} \|\mathbf{x}_m - \boldsymbol{\mu}_{k'}\|$.

In this heuristic, the only bit of randomness is the selection of the first data point. After that, it is completely deterministic (except in the rare case that there are multiple equidistant points in the second step). It is extremely important that when selecting the third mean, you select that point that maximizes the minimum distance to the closest other mean. You want the point that's as far away from all previous means as possible. The furthest-first heuristic is just that: a heuristic. It works very well in practice, though can be somewhat sensitive to outliers (which will often get selected as some

[4] *Smoothed Analysis of the k-Means Method*, Arthur, Manthey, Röglin: `http://wwwhome.math.utwente.nl/~mantheyb/journals/JACM_ArthurEA_kMeansSmoothed.pdf`

of the initial means). However, this outlier sensitivity is usually reduced after one iteration through the K-means algorithm. Despite being just a heuristic, it is quite useful in practice.
You can turn the heuristic into an algorithm by adding a bit more randomness. This is the idea of the **K-means++** algorithm, which is a simple randomized tweak on the furthest-first heuristic. The idea is that when you select the k^{th} mean, instead of choosing the absolute furthest data point, you choose a data point at random, with probability proportional to its distance squared.

K-means ++ algorithm (pseudocode)

1 $\boldsymbol{\mu}_1 \leftarrow \mathbf{x}_m$ - initialize first centroid, m is random;
2 for k = 2 to K do
3 $d_n \leftarrow \min_{k' < k} \|\mathbf{x}_n - \boldsymbol{\mu}_{k'}\|^2, \forall n$ - compute distances
4 $\mathbf{p} \leftarrow \frac{1}{\sum_n n_d}\mathbf{d}$ - normalize to probabilty distribution
5 m $\leftarrow$ random sample from $\mathbf{p}$ - pick an example at random
6 $\boldsymbol{\mu}_k \leftarrow \mathbf{x}_m$
7 end for
8 run K-means using $\boldsymbol{\mu}$ as initial centers

The advantage of the K-means ++ algorithm in comparison to the usual K-means algorithm is reflected in this theorem, that we state without proof:

Theorem: Let $\tilde{J}$ be the value of the loss function 2.1 obtained by running K-means++, and let $J^{(opt)}$ be the true global minimum. Then $\mathbb{E}[\tilde{J}] \leq 8(\log K + 2)J^{(opt)}$. Moreover, if the data is "well suited" for clustering, then $\mathbb{E}[\tilde{J}] \leq \mathcal{O}(1)J^{(opt)}$.
The notion of "well suited" for clustering informally states that the advantage of going from $K-1$ clusters to K clusters is "large." Formally, it means that $J_K^{(opt)} \leq \epsilon^2 J_{K-1}^{(opt)}$, where ϵ is the desired degree of approximation.

This theorem states that, with the help of the K++ algorithm, we will surely get "closer" to the global minimum of the loss function.

Of course, there is still the problem of choosing k, the number of clusters. Usually, you don't know beforehand how many clusters the data contains, and usually we cannot look at the data directly because it lies in a higher dimension than two or three. (Indeed, if you can look at your data and see obvious clusters like you can here, you may be better off clustering manually). So in practice, people often try different values of k and see how their results vary. We will discuss some other algorithms that help overcome this problem.

2.1.2 Gaussian Mixture Models

Using an algorithm such as K-Means leads to hard assignments, meaning that each point is definitively assigned a cluster center. The Euclidean distance approach classifies data into hard hyperspheres. This leads to some interesting problems: what if the true clusters actually overlap? What about data that is more spread out; how do we assign clusters then? What about data that cannot be separated by spheres? In the following picture we can see a dataset where clearly the K-means algorithm fails to separate the data into the two correct clusters:

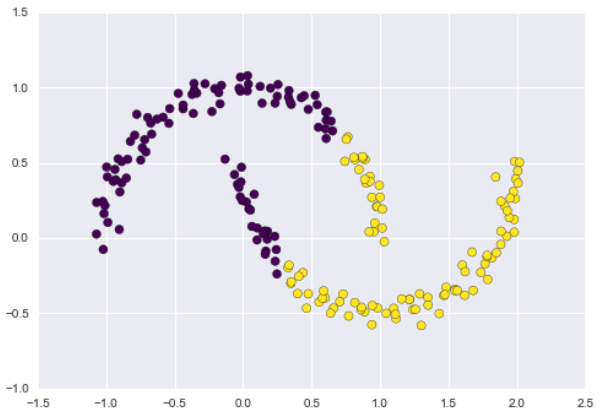


Figure 2.1: Clustering of data using the K-means algorithm. We see that K-means fails to classify the two moon-like datasets.

In order to soften the way that the K-means algorithm works and also allow for some flexibility, we can use so-called *mixture models*[5]. Mixture models allow us to give a distribution of data as a superposition of simpler distributions. One of the most used mixture models is the Gaussian mixture model (GMM). The idea behind Gaussian Mixture Models is to find the parameters of the Gaussians that best explain our data.

[5]Mixture model is a probabilistic model for representing the presence of sub-populations within an overall population, without requiring that an observed data set should identify the sub-population to which an individual observation belongs.

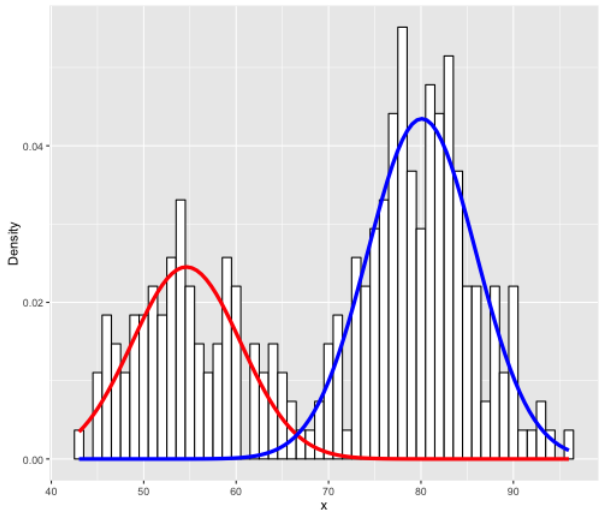


Figure 2.2: A distribution approximated as a superposition of two Gaussians

The motivation for Gaussian mixture models comes from the Central Limit Theorem, which tells us that enough random samples from *any* distribution will look like the normal distribution.
As a quick reminder, the multidimensional Gaussian distribution is:

$$\mathcal{N}(\boldsymbol{\mu},\, \boldsymbol{\Sigma}) = \det(2\pi\boldsymbol{\Sigma})^{-\frac{1}{2}}\, e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\intercal}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{2.3}$$

where $\boldsymbol{\Sigma}$ is the variance matrix, and $\boldsymbol{\mu}$ is the mean vector. The covariance matrix, in addition to telling us the variance of each dimension, also tells us the relationship between the inputs, i.e., if we change x, how does y tend to change?

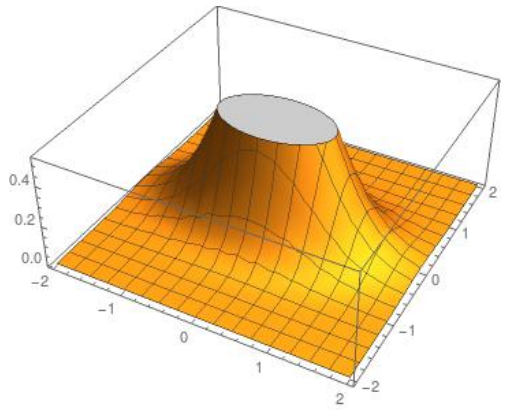


Figure 2.3: Section of a 2D elliptical Gaussian

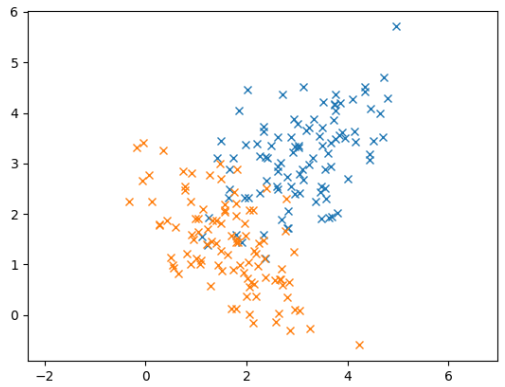


Figure 2.4: A dataset generated by two Gaussians

Additionally, K-Means does not take into account the covariance of our data. If we take a closer look at the Figure 2.4, the blue points seem to have a relationship between x and y: larger x values tend to produce larger y values. If we had two points that were equidistant from the center of the cluster, but one followed the trend and the other did not, K-Means would regard them as being equal, since it uses Euclidean distance. But it seems certainly more likely that the point that follows the trend should match closer to the Gaussian than the point that does not.

Since we know these data are Gaussian, let us try to fit Gaussians to them instead of a single cluster center.
This is sometimes called *generative modeling*. We are assuming that these data are Gaussian and we want to find parameters that maximize the likelihood of observing these data. In other words, we regard each point as being generated by a mixture of Gaussians and can compute that probability:

$$p(x) = \sum_{j=1}^{k} \phi_j \mathcal{N}(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \tag{2.4}$$

with the constraint

$$\sum_{j=1}^{k} \phi_j = 1 \tag{2.5}$$

The first equation tells us that a particular data point x is a linear combination of the k Gaussians. We weight each Gaussian with ϕ_j, which represents the strength of that Gaussian. The second equation is a constraint on

the weights: they all have to sum up to 1. We have three different parameters that we need to write update: the weights for each Gaussian ϕ_j, the means of the Gaussians $\boldsymbol{\mu}_j$, and the covariances of each Gaussian $\boldsymbol{\Sigma}_j$.

If we try to directly solve for the parameter, it turns out that we can actually find closed-forms, but we have to know the ϕ_j's. In other words, if we knew exactly which combination of Gaussians a particular point was taken from, then we could easily figure out the means and covariances. But this one critical flaw prevents us from solving GMMs using this direct technique. Instead, we have to come up with a better approach to estimate the weights, means, covariances. And actually we can do so, by using the EM algorithm.

The first part is the expectation step. In this step, we have to compute the probability that each data point was generated by each of the k Gaussians. In contrast to the K-Means hard assignments, these are called soft assignments since we're using probabilities. Note that we're not assigning each point to a Gaussian, we're simply determining the probability of a particular Gaussian generating a particular point. We compute this probability for a given Gaussian by computing $\phi_j \mathcal{N}(x; \mu_j, \Sigma_j)$ and normalizing by dividing by $\sum_{q=1}^{k} \phi_q \mathcal{N}(x; \mu_q, \Sigma_q)$. We are directly applying the Gaussian equation, but multiplying it by its weight ϕ_j. Then, to make it a probability, we normalize. In K-Means, the expectation step is analogous to assigning each point to a cluster. In the very first step, we assign random or uniformly distributed values for the parameters.

The second part is the maximization step. In this step, we need to update our weights, means, and covariances. Recall in K-Means, we simply took the mean of the set of points assigned to a cluster to be the new mean. We're going to do something similar here, except apply our expectations that we computed in the previous step. To update a weight ϕ_j, we simply sum up the probability that each point was generated by Gaussian j and divide by the total number of points. For a mean $\boldsymbol{\mu}_j$, we compute the mean of all points weighted by the probability of that point being generated by Gaussian j. For a covariance $\boldsymbol{\Sigma}_j$, we compute the covariance of all points weighted by the probability of that point being generated by Gaussian j. We do each of these for each Gaussian j. This way we have updated the weights, means, and covariances! In K-Means, the maximization step is analogous to moving the cluster centers.

Mathematically, at the expectation step, we are effectively computing a matrix where the rows are the data point and the columns are the Gaussians. An element at row i, column j is the probability that $x^{(i)}$ was generated by Gaussian j, namely:

$$W_j^{(i)} = \frac{\phi_j \mathcal{N}(x^{(i)}; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}{\sum_{q=1}^{k} \phi_q \mathcal{N}(x^{(i)}; \boldsymbol{\mu}_q, \boldsymbol{\Sigma}_q)}$$

The denominator just sums over all values to make each entry in W a probability. Now, we can apply the update rules.

$$\begin{aligned}
\phi_j &= \frac{1}{N} \sum_{i=1}^{N} W_j^{(i)} \\
\boldsymbol{\mu}_j &= \frac{\sum_{i=1}^{N} W_j^{(i)} x^{(i)}}{\sum_{i=1}^{N} W_j^{(i)}} \\
\boldsymbol{\Sigma}_j &= \frac{\sum_{i=1}^{N} W_j^{(i)} (x^{(i)} - \boldsymbol{\mu}_j)(x^{(i)} - \boldsymbol{\mu}_j)^T}{\sum_{i=1}^{N} W_j^{(i)}}
\end{aligned}$$

The first equation is just the sum of the probabilites of a particular Gaussian j divided by the number of points. In the second equation, we are just computing the mean, except we multiply by the probabilities for that cluster. Similarly, in the last equation, we are just computing the covariance, except we multiply by the probabilities for that cluster.

2.1.3 Python implementations

K-means clustering implementation

A nice application - Picture rendering

GMM

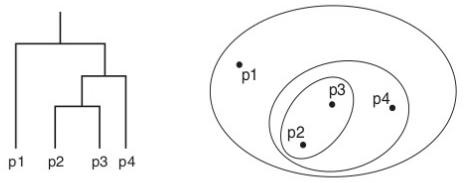
Will be discussed in class (I guess it doesn't make sense to write lines of code in LaTeX)

2.2 Hierarchical clustering

Hierarchical clustering techniques are a second important category of clustering methods. As with K-means, these approaches are relatively old compared to many clustering algorithms, but they still enjoy widespread use. There are two basic approaches for generating a hierarchical clustering:

- Agglomerative: Start with the points as individual clusters and, at each step, merge the closest pair of clusters. This requires defining a notion of cluster proximity.
- Divisive: Start with one, all-inclusive cluster and, at each step, split a cluster until only singleton clusters of individual points remain. In this case, we need to decide which cluster to split at each step and how to do the splitting.

Agglomerative hierarchical clustering techniques are by far the most common, and, in this section, we will focus exclusively on these methods. A hierarchical clustering is often displayed graphically using a tree-like diagram called a *dendrogram*, which displays both the cluster relationships and the order in which the clusters were merged (agglomerative view) or split (divisive view). For sets of two-dimensional points, such as those that we will use as examples, a hierarchical clustering can also be graphically represented using a nested cluster diagram. Figure 2.5 shows an example of these two types of figures for a set of four two-dimensional points. These points were clustered using the *single-link* technique, which we will discuss in the next subsection.



(a) Dendrogram. (b) Nested cluster diagram.

Figure 2.5: Hierarchical clustering

2.2.1 Basic Agglomerative Hierarchical Clustering Algorithm

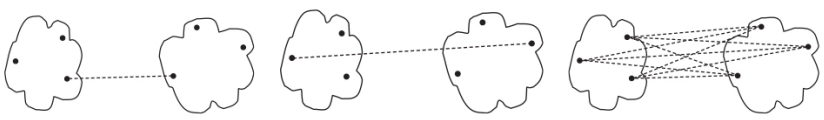
Many agglomerative hierarchical clustering techniques are variations on a single approach: starting with individual points as clusters, successively merge

the two closest clusters until only one cluster remains. This approach is expressed more formally in the algorithm below.

```
Basic agglomerative hierarchical clustering algorithm
1:  Compute the proximity matrix, if necessary.
2:  repeat
3:  Merge the closest two clusters.
4:  Update the proximity matrix to reflect the proximity
    between the new cluster and the original clusters.
5:  until Only one cluster remains.
```

2.2.2 Defining Proximity between Clusters

The key operation of the algorithm above is the computation of the proximity between two clusters, and it is the definition of cluster proximity that differentiates the various agglomerative hierarchical techniques that we will discuss. Cluster proximity is typically defined with a particular type of cluster in mind. For example, many agglomerative hierarchical clustering techniques, such as MIN, MAX, and Group Average, come from a graph-based view of clusters. MIN defines cluster proximity as the proximity between the closest two points that are in different clusters. Alternatively, MAX takes the proximity between the farthest two points in different clusters to be the cluster proximity (If our proximities are distances, then the names, MIN and MAX, are short and suggestive. For similarities, however, where higher values indicate closer points, the names seem reversed. For that reason, we usually prefer to use the alternative names, single link and complete link, respectively.) Another graph-based approach, the group average technique, defines cluster proximity to be the average pairwise proximities (average length of edges) of all pairs of points from different clusters. We illustrate these three approaches in the following figure:



(a) MIN (single link.) (b) MAX (complete link.) (c) Group average.

Figure 2.6: Different proximities that we can define between clusters

2.2.3 Group Average

For the group average version of hierarchical clustering, the proximity of two clusters is defined as the average pairwise proximity among all pairs of points in the different clusters. This is an intermediate approach between the single and complete link approaches. Thus, for group average, the cluster proximity (C_i, C_j) of clusters C_i and C_j, which are of size m_i and m_j, respectively, is expressed by the following equation:

$$Proximity(C_i, C_j) = \frac{\sum_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} proximity(\mathbf{x}, \mathbf{y})}{m_i \times m_j}$$

2.2.4 Ward's method

If, instead, we take a prototype-based view, in which each cluster is represented by a centroid, different definitions of cluster proximity are more natural. When using centroids, the cluster proximity is commonly defined as the proximity between cluster centroids. An alternative technique, Ward's[6] method, also assumes that a cluster is represented by its centroid, but it measures the proximity between two clusters in terms of the increase in the Euclidean distance that results from merging the two clusters. Like K-means, Ward's method attempts to minimize the sum of the squared distances of points from their cluster centroids.

2.2.5 An instructive example

Blackboard

2.3 DBSCAN

Now we turn to another category of clustering, called *density-based clustering*. DBSCAN stands for Density-based spatial clustering of applications with noise. We will first talk about density, and then see how does the DBSCAN algorithm work.

Traditional Density: Center-Based Approach

Although there are not as many approaches for defining density as there are for defining similarity, there are several distinct methods. In this section we dis- cuss the center-based approach on which DBSCAN is based. In the

[6]For physicists: No, not that Ward from QFT.

center-based approach, density is estimated for a particular point in the data set by counting the number of points within a specified radius, ϵ, of that point. This includes the point itself. This method is simple to implement, but the density of any point will depend on the specified radius.
The center-based approach to density allows us to classify a point as being

- in the interior of a dense region (a core point)
- on the edge of a dense region (a border point), or
- in a sparsely occupied region (a noise or background point).

Figure 2.8 graphically illustrates the concepts of core, border, and noise points.
Core points: These points are in the interior of a density-based cluster. A point is a core point if the number of points within a given neighborhood around the point as determined by the distance function and a user-specified distance parameter, ϵ, exceeds a certain threshold, `MinPts`, which is also a user-specified parameter. In Figure 2.7, point A is a core point, for the indicated ϵ if `MinPts` ≤ 7.
Border points: A border point is not a core point, but falls within the neighborhood of a core point. In Figure 2.8, point B is a border point. A border point can fall within the neighborhoods of several core points.
Noise points: A noise point is any point that is neither a core point nor a border point. In Figure 2.8, point C is a noise point.

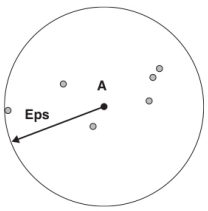


Figure 2.7: A ball of radius ϵ

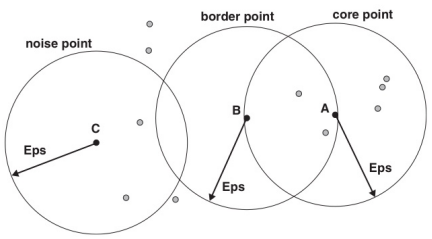


Figure 2.8: Core, border and noise point

2.3.1 The DBSCAN algorithm

Given the previous definitions of core points, border points, and noise points, the DBSCAN algorithm can be informally described as follows. Any two core points that are close enough—within a distance ϵ of one another—are put in the same cluster. Likewise, any border point that is close enough to a core point is put in the same cluster as the core point. (Ties may need to be resolved if a border point is close to core points from different clusters.) Noise points are discarded. The formal details are given in the algorithm below. This algorithm uses the same concepts and finds the same clusters as the original DBSCAN, but is optimized for simplicity, not efficiency.

```
    DBSCAN algorithm
1:  Label all points as core, border, or noise points.
2:  Eliminate noise points.
3:  Put an edge between all core points that are within Eps
    of each other.
4:  Make each group of connected core points into a separate
    cluster.
5:  Assign each border point to one of the clusters of its
    associated core points.
```

DBSCAN can find many clusters that could not be found using K-means. However, DBSCAN has trouble when the clusters have widely varying densities. It also has trouble with high-dimensional data because density is more difficult to define for such data. Finally, DBSCAN can be expensive when the computation of nearest neighbors requires computing all pairwise proximities, as is usually the case for high-dimensional data. Here is an example of the DBSCAN algorithm in action.

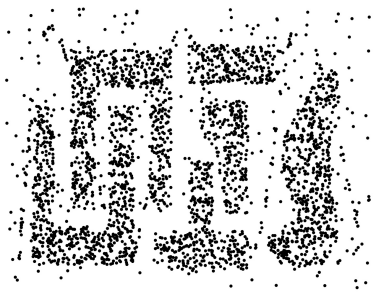
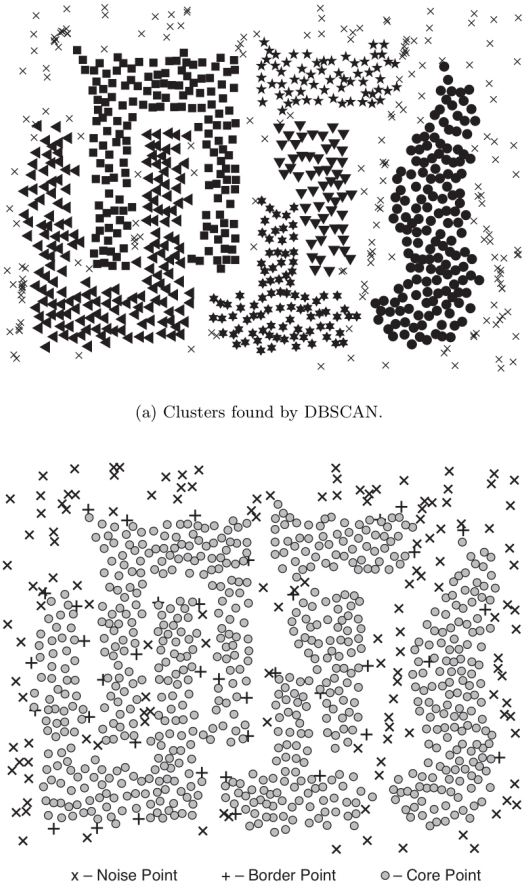


Figure 2.9: Sample data



(a) Clusters found by DBSCAN.

Figure 2.10: Cluster analysis with the help of the DBSCAN algorithm

Chapter 3

Anomaly detection

In data mining, anomaly detection (also outlier detection) is the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data. Typically the anomalous items will translate to some kind of problem such as bank fraud, a structural defect, medical problems or errors in a text. Anomalies are also referred to as outliers, novelties, noise, deviations and exceptions.
In particular, in the context of abuse and network intrusion detection, the interesting objects are often not rare objects, but unexpected bursts in activity. This pattern does not adhere to the common statistical definition of an outlier as a rare object, and many outlier detection methods (in particular unsupervised methods) will fail on such data, unless it has been aggregated appropriately. Instead, a cluster analysis algorithm may be able to detect the micro clusters formed by these patterns.

3.1 Local Outlier Factor - LOF

We will discuss one of the most used algorithms for anomaly detection, namely the Local Outlier Factor algorithm. The local outlier factor is based on a concept of a local density, where locality is given by k nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors (similarly as in the DBSCAN algorithm). These are considered to be outliers.

The local density is estimated by the typical distance at which a point can be reached from its neighbors. The definition of *reachability distance* used in LOF is an additional measure to produce more stable results within

clusters.

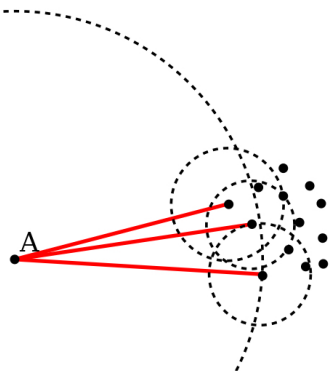


Figure 3.1: Basic idea of LOF: comparing the local density of a point with the densities of its neighbors. A has a much lower density than its neighbors.

Let k-distance(A) be the distance of the object A to the k-th nearest neighbor. Note that the set of the k nearest neighbors includes all objects at this distance, which can in the case of a "tie" be more than k objects. We denote the set of k nearest neighbors as $N_k(A)$. This distance is used to define what is called reachability distance:

$$\text{reachability-distance}_k(A, B) = \max\{\text{k-distance}(B), d(A, B)\}$$

In words, the reachability distance of an object A from B is the true distance of the two objects, but at least the k-distance of B. Objects that belong to the k nearest neighbors of B (the *core* of B, as we have defined while analyzing the DBSCAN algorithm) are considered to be equally distant. The reason for this distance is to get more stable results. Note that this is not a distance in the mathematical definition, since it is not symmetric (While it is a common mistake to always use the k-distance, this yields a slightly different method, referred to as *Simplified LOF*).

The *local reachability density* of an object A is defined by

$$\text{lrd}_k(A) := 1/\left(\frac{\sum_{B \in N_k(A)} \text{reachability-distance}_k(A, B)}{|N_k(A)|}\right)$$

which is the inverse of the average reachability distance of the object A from its neighbors. Note that it is not the average reachability of the neighbors from A (which by definition would be the k-distance(A)), but the

distance at which A can be "reached" from its neighbors. With duplicate points, this value can become infinite.

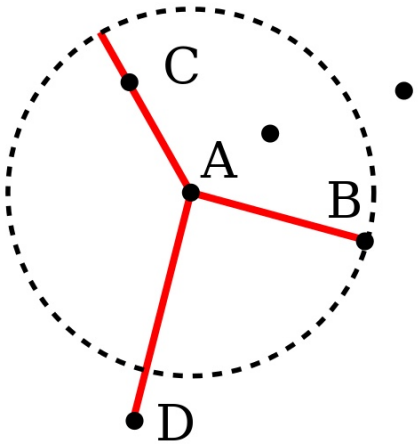


Figure 3.2: Illustration of the reachability distance. Objects B and C have the same reachability distance (k=3), while D is not a k nearest neighbor

The local reachability densities are then compared with those of the neighbors using:

$$\mathrm{LOF}_k(A) := \frac{\sum_{B \in N_k(A)} \frac{\mathrm{lrd}(B)}{\mathrm{lrd}(A)}}{|N_k(A)|} = \frac{\sum_{B \in N_k(A)} \mathrm{lrd}(B)}{|N_k(A)|} / \mathrm{lrd}(A)$$

which is the average local reachability density of the neighbors divided by the object's own local reachability density. A value of approximately 1 indicates that the object is comparable to its neighbors (and thus not an outlier). A value below 1 indicates a denser region (which would be an inlier), while values significantly larger than 1 indicate outliers.

3.1.1 Python implementation

`https://scikit-learn.org/stable/auto_examples/neighbors/plot_lof_outlier_detection.html`