Lecture 16
Unsupervised Learning

Blake VanBerlo

March 6, 2023

Contents

1 Learning Goals

By the end of the lecture, you should be able to

- Explain the main tasks in unsupervised learning.
- Describe and trace the steps of the k-means algorithm.
- Apply strategies for selecting the number of clusters for k-means.
- Describe autoencoders and how they are trained.
- Describe generative adversarial networks and how they are trained.

2 Introduction to Unsupervised Learning

Recall from the previous lectures that there are three major types of machine learning: supervised learning, unsupervised learning, and reinforcement learning. We saw that supervised learning algorithms are trained to predict outputs from inputs in situations where we have access to ground truth targets that correspond to the inputs.

In unsupervised learning problems, we do not have access to labelled input-output examples. Instead of learning to predict correct outputs, the goal is to uncover underlying structure in an unlabelled dataset of training examples.

There are two major types of tasks of unsupervised learning tasks:

1. **Representation Learning**: Learning low-dimensional representations of examples; the goal is to find a simpler set of features that sufficiently describe the examples
2. **Generative Modelling**: Learning a probability distribution from which new examples can be drawn as samples

Several unsupervised learning algorithms are capable of performing both types of tasks.

A common task in unsupervised learning is *clustering*. In general, clustering is defined as the task of learning how to group training examples into categories called *clusters* (or classes), making it a simple form of representation learning. For example, given a million images that either contain a cat or a dog, a clustering algorithm would ideally learn to sort the images into two clusters: one for cats and one for dogs. The clustering should minimize some error function on the cluster predictions for the examples. We will only consider the clustering task in our discussion of unsupervised learning.

The clustering task can be further decomposed into two types:

1. **Hard clustering:** Every example is assigned to one cluster with certainty. *k-means* is an example of a hard clustering algorithm.
2. **Soft clustering:** Every example has a probability distribution over all possible clusters. The *expectation maximization* algorithm combined with a *naive Bayes classifier* is an example of a soft clustering algorithm.

3 k-Means

The k-means algorithm is a hard clustering algorithm. It learns to assign examples to definite classes. The input to the algorithm is the number of clusters (i.e. the value of k), and the training examples. The goal of the algorithm is to learn a representation of the dataset that assigns examples to the appropriate class, denoted as classes $1, 2, ..., k$.

3.1 The k-means Algorithm

Suppose that we have a set of training examples X. For each $x \in X$, we would like to learn a clustering that assigns x to a class $c \in \{1, 2, ..., k\}$. Suppose that each training example x is composed of n features, i.e. $x = \langle x_1, x_2, ..., x_n \rangle$. Note that each feature is real-valued — $x_i \in \mathbb{R}$. The k-means algorithm can therefore be applied to cluster data of several types, such as coordinates, rows of tabular data, images, or even audio waveforms.

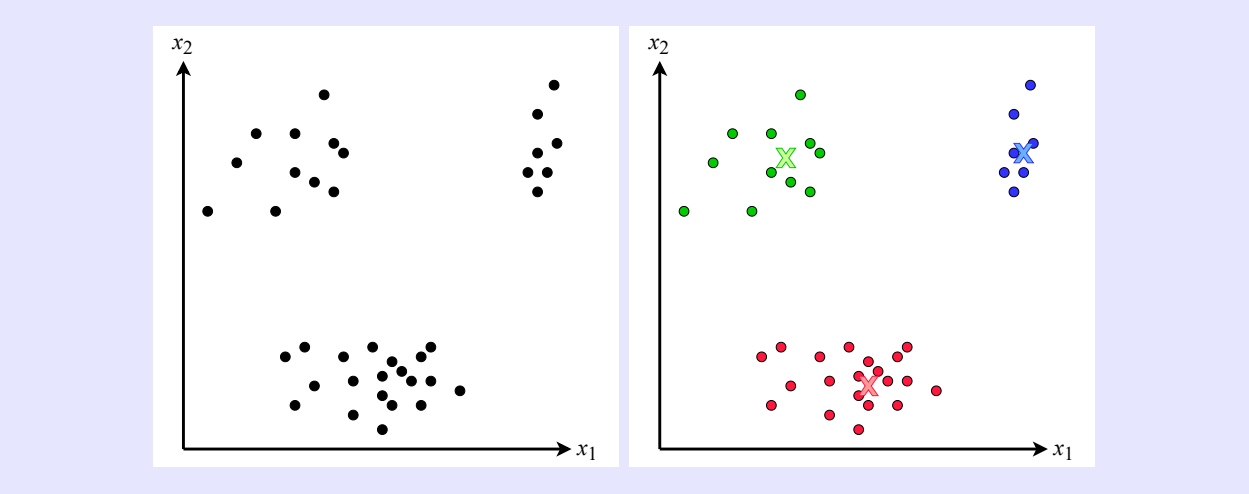
k-means learns a *centroid* for each cluster and assigns examples to the cluster whose centroid is closest to the example. Centroids have the same number of features as the training examples. By the "closest centroid", we mean centroid with the smallest distance from the example. Distance is defined by the distance function $d(c, x)$ between centroid c and example x. The Euclidean distance (i.e. L_2 norm), shown below, is a commonly used distance function for k-means.

$$d(c, x) = \sqrt{\sum_{j=1}^{n} (c_j - x_j)^2}$$

To gain an intuition of what k-means is trying to accomplish, consider the example below.

Example:

The left image gives an example of a small and simple dataset where the examples have 2 features: x_1 and x_2. If we apply k-means clustering with $k = 3$, we expect to obtain a clustering similar to the right image. The output of the clustering algorithm is the 3 centroids corresponding to the centres of each cluster (indicated by the coloured X's). Each example is a part of the cluster whose centroid is the closest.



The algorithm alternates between two major steps:

1. *Centroid update:* Set the centroid of each cluster as the feature-wise mean of each example currently assigned to the cluster.
2. *Cluster assignment:* Assign each training example x to the cluster with the closest centroid.

We are now ready to discuss the full k-means clustering algorithm, which is given below.

Algorithm 1 The k-Means Clustering Algorithm

inputs: $X \in \mathbb{R}^{m \times n}$, ▷ m examples with n features
$k \in \mathbb{N}$, ▷ # of clusters
$d(c, x)$ ▷ Distance function for comparing examples to centroids
$Y \in \mathbb{N}^m \leftarrow$ Random cluster assignment for each example
$C \in \mathbb{R}^{k \times n} \leftarrow$ Random centroid feature values
converged $\leftarrow False$
while not converged **do**
  for $c = 1, 2, ..., k$ **do**
    $X_c \leftarrow$ set of examples in class c
    $n_c \leftarrow$ # of examples in class c
    $C[c] \leftarrow \frac{1}{n_c} \sum_{i=1}^{n_c} X_c[i]$ ▷ Calculate the centroid for cluster c
  $Y' \leftarrow Y$
  for $i = 1, 2, ..., m$ **do**
    $Y[i] \leftarrow \arg\min_c d(C[c], X[i])$ ▷ Assign each example to a cluster
  converged $\leftarrow Y == Y'$
return Y

Let's go through the pseudocode provided above. First, we randomly assign each of the m examples in the training set to a class $c \in [1, k]$, keeping track of it in Y. We then randomly initialize k cluster centroids. Note that, like each training example, each centroid has n real-valued features.

Following this, we alternate between the two major steps of k-means until the cluster assignments converge. First, we calculate new centroids. To calculate the new value of centroid c, we take the feature-wise mean of each example that is currently assigned to cluster c. Once we have the new clusters, we reassign each example in the training set to the cluster that minimizes the distance between the example and that cluster's centroid. If the new cluster assignments are the same as the cluster assignments at the beginning of the iteration, the algorithm terminates, returning the cluster assignments Y. Otherwise, another iteration is performed.

3.2 Example of k-means

Example: Let's perform 1 iteration of k-means. We will use a small dataset that has 3 features per example, with $k = 2$ and Euclidean distance. The left table below is the dataset and the right table are initial centroid values for each cluster. Use the initial cluster assignment $Y = [1, 1, 2, 2]$.

Example	x_1,	x_2	x_3
1	0.2	0.5	0
2	-0.6	2.1	1.2
3	-0.5	1.9	1.3
4	0.1	0.5	-0.3

k	c_1,	c_2	c_3
1	0.3	0.8	-0.5
2	-0.1	-0.5	1.0

Calculate new centroids.

For centroid 1:
$X_1 \leftarrow$ Examples 1 and 2
$n_1 \leftarrow 2$
$C_1 \leftarrow \frac{1}{2}\Big(\langle 0.2, 0.5, 0\rangle + \langle -0.6, 2.1, 1.2\rangle\Big) = \langle -0.2, 1.3, 0.6\rangle$

For centroid 2:
$X_2 \leftarrow$ Examples 3 and 4
$n_2 \leftarrow 2$
$C_2 \leftarrow \frac{1}{2}\Big(\langle -0.5, 1.9, 1.3\rangle + \langle 0.1, 0.5, -0.3\rangle\Big) = \langle -0.2, 1.2, 0.5\rangle$

Now update example assignments. First set $Y' \leftarrow Y$.

$$Y[1] \leftarrow \arg\min_c \left(\sqrt{(-0.2-0.2)^2 + (1.3-0.5)^2 + (0.6-0)^2}, \sqrt{(-0.2-0.2)^2 + (1.2-0.5)^2 + (0.5-0)^2} \right)$$
$$= 2$$
$$Y[2] \leftarrow \arg\min_c \left(\sqrt{(-0.2-(-0.6))^2 + (1.3-2.1)^2 + (0.6-1.2)^2}, \sqrt{(-0.2-(-0.6))^2 + (1.2-2.1)^2 + (0.5-1.2)^2} \right)$$
$$= 1$$
$$Y[3] \leftarrow \arg\min_c \left(\sqrt{(-0.2-(-0.5))^2 + (1.3-1.9)^2 + (0.6-1.3)^2}, \sqrt{(-0.2-(-0.5))^2 + (1.2-1.9)^2 + (0.5-1.3)^2} \right)$$
$$= 1$$
$$Y[4] \leftarrow \arg\min_c \left(\sqrt{(-0.2-0.1)^2 + (1.3-0.5)^2 + (0.6-(-0.3))^2}, \sqrt{(-0.2-0.1)^2 + (1.2-0.5)^2 + (0.5-(-0.3))^2} \right)$$
$$= 2$$

Check for convergence. Since $Y \neq Y'$, we would now start another iteration using the updated cluster centroids and assignments.

3.3 Finding the best solution

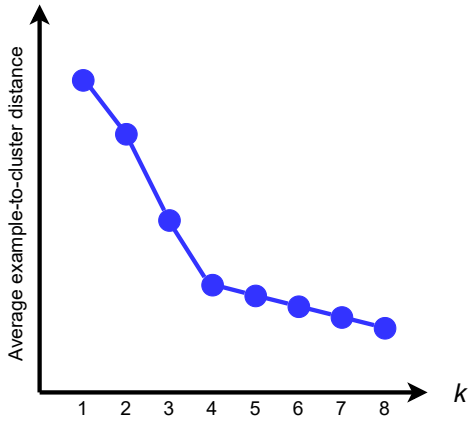
k-means is guaranteed to converge. However, the solution is not guaranteed to be optimal. To increase the chances of finding a better solution, we could run k-means multiple times with different initial random cluster assignments.

Another way to ensure that k-means finds a solution that sufficiently incorporates information from each feature is to scale the features such that their domains are similar. k-means is rather sensitive to the relative scales of each feature. For example, consider examples whose features are basic physical characteristics of humans. Height (in centimetres) is distributed over a different scale than age (in years).

3.4 Choosing a proper k

The choice of the number of clusters (k) greatly determines the outcome of the clustering. As long as there are less than or equal to $k+1$ examples, running k-means with $k+1$ clusters will result in lower error than with k clusters. A couple of strategies for selecting k are discussed below.

1. Elbow Method: If you execute k-means for multiple values of $k = 1, 2, ..., k_{max}$, we can plot the average distance across all examples and their assigned cluster centroids. Such a plot may resemble the example below.



We can select a reasonable k by identifying the *elbow point* of this plot; that is, the value of k after which there is a drastic reduction in error improvement. For the plot above, $k = 4$ is the elbow point.

The elbow method can be ambiguous, since it is conducted by manually inspecting the plot and identifying the elbow point. In some cases, it may be hard to discern the elbow point.

2. Silhouette Analysis: Again, we execute k-means for multiple values of $k = 1, 2, ..., k_{max}$. We then calculate the average silhouette score for each k. We select the value of k that maximizes the average silhouette score across the dataset. An advantage of Silhouette Analysis is that is more objective than the Elbow Method.

   Given a previously trained clustering with k clusters, the *silhouette score* of example $s(x) \in [-1, 1]$ indicates how well the example fits within its cluster and how poor it fits into other clusters. Higher values are better.

   The silhouette score is calculated as follows. Let C_i be the cluster that example x is in. We define the following quantities:

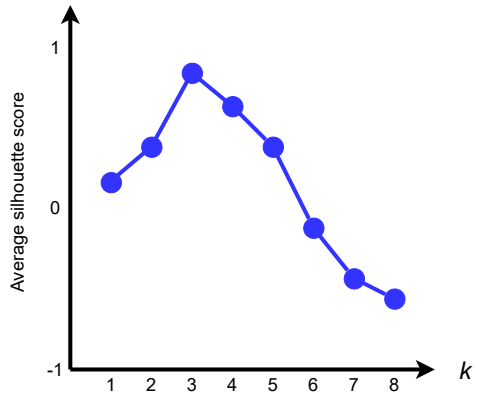
$$a(x) = \frac{1}{|C_i| - 1} \sum_{x' \in C_i, x' \neq x} d(x, x')$$

$$b(x) = \min_{j \neq i} \frac{1}{C_j} \sum_{x' \in C_j} d(x, x')$$

   Above, $a(x)$ is the average distance from examples x to all other examples in its own cluster, C_i. $b(x)$ is the smallest of the average distance of x to examples in any other cluster. Putting it together, we get the silhouette score $s(x)$:

$$s(x) = \begin{cases} \frac{b(x) - a(x)}{\max(a(x), b(x))} & \text{if } |C_x| > 1 \\ 0 & \text{if } |C_x| = 1 \end{cases}$$

Once all silhouette scores are calculated for each example for each cluster, calculate the average silhouette score across the dataset for each value of k. Choose the k that maximizes the average silhouette score. For example, $k = 3$ would be the best choice for the number of clusters, given the below plot.



4 Autoencoders

Autoencoders are a type of unsupervised model that seek to learn a low-dimensional representation of examples. In other words, they aim to learn a mapping from input examples x to a smaller representation z. Autoencoders consists of two main parts:

1. *Encoder* $e(x)$: a function that maps x to a low-dimensional representation $\hat{z}$
2. *Decoder* $d(\hat{z})$: a function that maps $\hat{z}$ to its original representation x

Putting it together, an autoencoder implements $\hat{x} = d(e(x))$, where $\hat{x}$ is the *reconstruction* of the original input x. Ideally, the decoder inverts the encoder's operation, resulting in successful reconstruction of the input. The encoder and decoder are learned such that the low-dimensional representation $\hat{z}$ contains as much information about x as needed to reconstruct it.

Autoencoders are often trained to minimize the mean squared error function, which gives the sum of squares of the differences between the features of the original input example and the prediction:

$$E = \sum_i (x_i - g(f(x_i)))^2$$

4.1 Linear Autoencoder

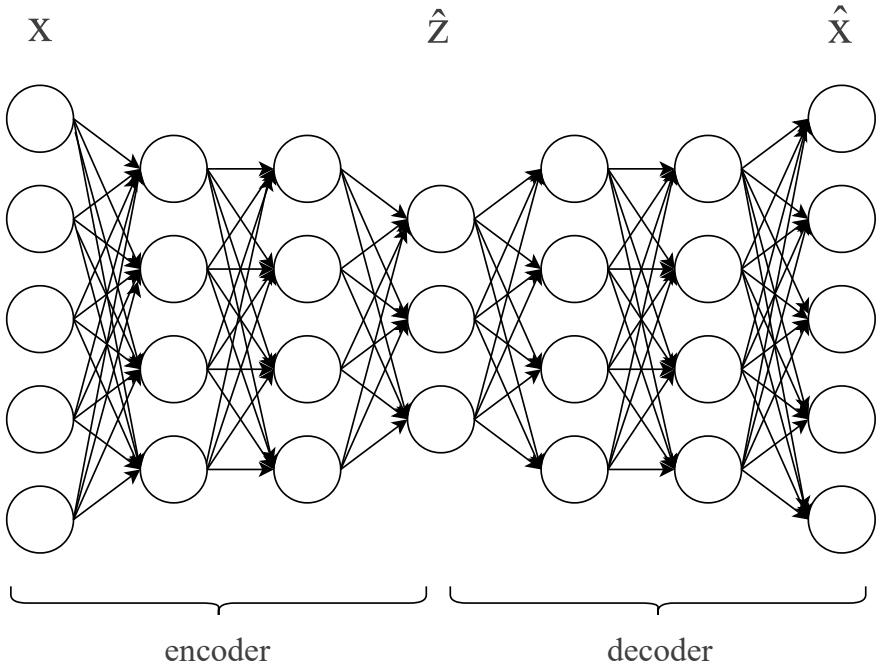
The linear autoencoder is the simplest form of autoencoder. e and d are linear functions that use a shared weight matrix W, such that:

$$\hat{z} = Wx$$
$$\hat{x} = W^\top \hat{z}$$

With $W^\top$ as the left inverse of W, the encoder and decoder can share the same weights. A properly trained W should span the principal components of the examples in the dataset.

4.2 Deep Neural Network Autoencoders

For complex examples, an autoencoder can be realized using a deep neural network. In this case, both the encoder $e(x)$ and decoder $d(\hat{z})$ are separate neural networks. The following illustration provides an example of an encoder architecture. The encoder and decoder are both 3-layer feedforward neural networks. The entire network consists of the decoder connected to the end of the encoder. The network's weights can be learned via backpropogation to minimize an error function that penalizes differences between the input and the reconstruction (e.g. mean squred error).



5 Generative Adversarial Networks

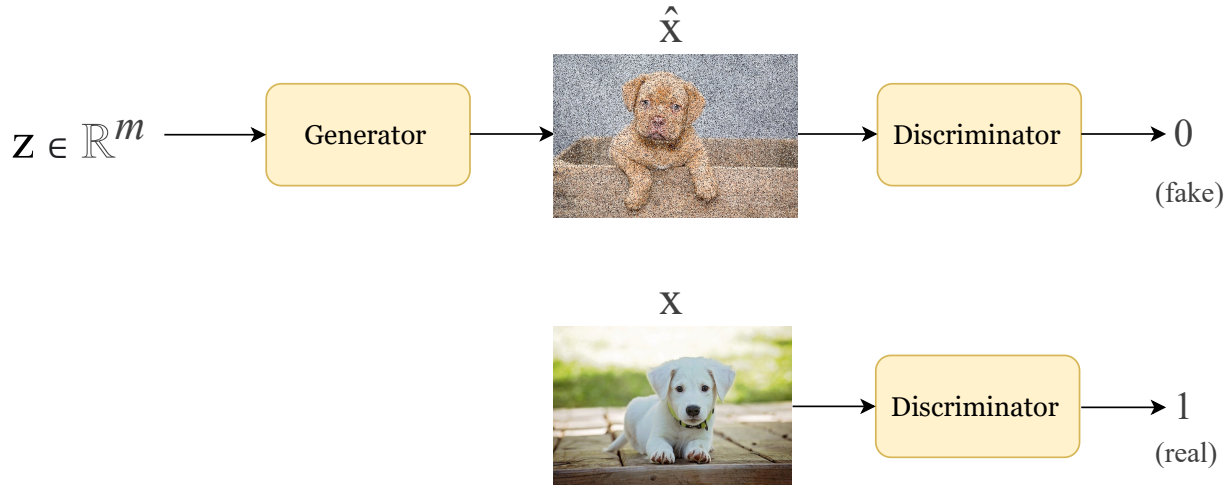
Generative adversarial networks are an example of a generative unsupervised learning algorithm. A generative adversarial network (GAN) consists of a pair of neural networks:

- *Generator* $g(z)$: Given a vector z in latent space, the generator produces an example $\hat{x}$ that is drawn from a probability distribution that approximates the true distribution of training examples. z is typically sampled from a Gaussian distribution.
- *Discriminator* $d(x)$: A classifier that predicts whether an example x is a real example in the training set or a fake example created by the generator

The generator in a GAN and the decoder in an autoencoder are similar in that both models seek to produce outputs that are in the same distribution as the training inputs. The main

difference lies in that the generator should have learned about the distribution of training examples. As a result, it should produce distinct examples that are different than the training examples, but similar enough such that they could believably be training examples.

The illustration below exhibits the intended functionality of a generator and discriminator. Suppose you are training a GAN to produce realistic images of dogs. The top of the illustration depict the generator producing an image of a dog from a randomly sampled latent vector z. The discriminator correctly classifies the generated image as "fake". On the bottom of the image, the discriminator correctly classifies the generated image as "real". By convention, the targets 0 and 1 correspond to fake and real examples respectively.



GANs can be trained with a minimax error function:

$$\mathbb{E}_x[\log(d(x))] + \mathbb{E}_z[\log(1 - d(g(z)))]$$

The generator and discriminator optimize this objective in different ways.

- The *discriminator* attempts to maximize this function. The first term is maximized when the discriminator is able to correctly classify real examples. The second term is maximized when the discriminator correctly classifies fake examples created by the generator.
- The *generator* attempts to minimize this function. Although it has no control over the first term, minimizing the second term corresponds to fooling the discriminator.

When training converges, the generator should be producing realistic images. Accordingly, the discriminator should output $\frac{1}{2}$, indicating that it is maximally uncertain whether the examples it sees are real or fake. The end result is a generative model that can produce novel instances that resemble examples in the training set.