Clustering: Introduction

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

```
In [ ]: %reload_ext autoreload
%autoreload 2
%matplotlib inline

In [ ]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [ ]: from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

Clustering

- **Clustering** is the process of grouping objects according to some distance measure
- The goals:
  - two objects in the same cluster are a small distance from each other
  - two objects in different clusters are a large distance from each other
- E.g. how would you cluster these dogs?
- Applications:
  - Genetics: discovering groups of genes that express themselves in similar ways
  - Marketing: segmenting customers for targeted advertising or to drive new product development
  - Social network analysis: discovering communities in social networks
  - Social sciences: analysing populations based on demographics, behaviour, etc
  - Genetic algorithms: identifying population niches in an effort to maintain diversity
- Note: Clustering algorithms assign the objects to groups, but they are typically not capable of giving meaningful labels (names) to the groups
Clustering algorithms

- There are many, many algorithms, falling roughly into two kinds:
  - **Point-assignment algorithms:**
    - objects are initially assigned to clusters, e.g., arbitrarily
    - then, repeatedly, each object is re-considered: it may be assigned to a cluster to which it is more closely related
  - **Hierarchical algorithms:** produce a tree of clusters
    - **Agglomerative algorithms** ('bottom-up'):
      - each object starts in a 'cluster' on its own;
      - then, recursively, pairs of clusters are merged to form a parent cluster
    - **Divisive algorithms** ('top-down'):
      - all objects start in a single cluster;
      - then, recursively, a cluster is split into child clusters
- There are lots of other ways of distinguishing clustering algorithms from each other, e.g.
  - partitioning: must every object belong to exactly one cluster, or may some objects belong to more than one cluster and may some objects belong to no cluster?
  - hard vs. soft: is membership of a cluster Boolean (an object belongs to a cluster or it does not) or is it fuzzy (there are degrees of membership, e.g. it is 0.8 true that this object belongs to this cluster) or probabilistic
  - whether they only work for certain distance measures (e.g. Euclidean, Manhattan, Chebyshev) and not for others (e.g. cosine)
  - whether they assume a dataset that fits into main memory or whether they scale to larger datasets
  - whether they assume all the data is available up-front, or whether they assume it arrives over time
- We'll study two of the simpler algorithms: one point-assignment and one hierarchical

**k**-Means Clustering

- The **k-means algorithm** is the best-known point-assignment algorithm
  - E.g. the KMeans class in scikit-learn
- It assumes that you know the number of clusters, *k*, in advance
- Given a dataset of examples (as vectors) $X$ it returns a partition of $X$ into *k* subsets
- Key concept: the centroid of a cluster
  - the mean of the examples in that cluster, i.e. the mean of each feature
Centroids

- Class exercise: What are the centroids of these clusters?
  
  1. \(
  \begin{bmatrix}
  1 \\
  1 \\
  1 \\
  4 \\
  2 \\
  6 \\
  11 \\
  \end{bmatrix}
  \) 

  2. \{3\} 

  3. \{2,\begin{bmatrix}4\end{bmatrix}\} 

- Observations:
  - The centroid of a cluster that contains just one example is the example itself
  - The centroid of a cluster that contains more than one example may not even be one of the examples in the cluster

\(k\)-Means Algorithm

- It starts by choosing \(k\) examples from \(X\) to be the initial centroids, e.g. randomly
- Then, repeatedly,
  - Assignment step: Each example \(x \in X\) is assigned to one of the clusters: the one whose centroid is closest to \(x\)
  - Update step: It re-computes the centroids of the clusters
When to stop?

- If you run it for enough iterations, there usually comes a point when
  - In the update step, the centroids don’t change
  - Hence, in the assignment step, the clustering doesn’t change
- But there is a small risk that it never happens and that the algorithm oscillates between two or more equally good solutions
- Therefore, most implementations have a maximum number of iterations (max_iter in scikit-learn)
- They might stop earlier, when the algorithm converges — next slide
Inertia and Convergence

- What is $k$-means trying to achieve?
  - A clustering that minimizes inertia: the within-cluster sum of distances
  - I.e. the sum of the distances from each $x \in X$ to its centroid is as low as possible
  (Advanced: The algorithm is more correctly formalized as trying to minimise the within-cluster sum of squares of distances but, with Euclidean distance, the best clustering is the same)
- If you run it for enough iterations, it will converge
  - I.e. the inertia will remain unchanged between iterations
  The algorithm can stop at this point
- Most implementations have a tolerance (tol in scikit-learn):
  - They stop when the change in inertia falls below the tolerance, rather than waiting for zero change

Local and Global Minima

- Even if the algorithm converges (no improvement in inertia), the clustering it converges on might not be the global minimum (the one with lowest possible inertia)
- $k$-means produces different clustering depending on the choice of the initial $k$ centroids
- For a given set of initial centroids, the clustering it converges on might be a local minimum:
  - For these initial centroids, no better clustering can be found, but it's not the very best clustering possible
- Class exercise. Here, $X$ contains four examples at the corners of a rectangle:

  ![Diagram of four examples at corners of a rectangle]

  - For $k = 2$, choose initial centroids that result in a global minimum
  - And choose $k = 2$ centroids that give a local minimum
  - Let's look at ways of reducing the problem...

Avoiding local minima: re-running

- The obvious solution is to run $k$-means multiple times (with different initial centroids) and return the clustering that has the lowest inertia
- No guarantee of finding the global minimum this way but likely to be better
- E.g. scikit-learn the number of runs (n_init) is 10, by default
Avoiding local minima: better initial centroids

- Choosing the initial $k$ centroids at random from $X$ has problems:
  - The algorithm can return different clusters for $X$ each time it is run
  - The clustering it returns may be a local minima
  - A poor choice can increase the number of iterations needed for convergence
- There are many alternatives to choosing wholly randomly, e.g.:
  - Insert into $\text{Centroids}$ an example $x \in X$, chosen at random with uniform probability
  - While $|\text{Centroids}| < k$
    - Insert into $\text{Centroids}$ a different example $x \in X$, chosen with probability proportional to $(\min_{x' \in \text{Centroids}} \text{dist}(x, x'))^2$
- $k$-means++ is the name of the $k$-means algorithm when using the above method
  - It still has randomness, so it still suffers from the problems above, but typically less so
- In scikit-learn, the $\text{init}$ parameter can have values ‘$k$-means++’ (default) or ‘random’

$k$-means clustering: discussion

- $k$-means can work well
  - But not so much in the presence of outliers or when the natural clusters are elongated or irregular shapes
- The curse of dimensionality may be relevant
  - You might want to do dimensionality reduction, e.g. using PCA, first
- The algorithm mostly scales well to larger data
  - There are variants for speed-up, e.g. $\text{MiniBatchKMeans}$ in scikit-learn
- There is the problem of choosing $k$ in advance
  - Why does it not make sense to run it with all $k$ in $[2, m]$ and choose the clustering with lowest inertia?
  - There are point-assignment algorithms that do not require you to choose $k$ in advance

$k$-Means in scikit-learn

In [ ]:
```python
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")
```

In [ ]:
```python
# The features we want to select
numeric_features = ["flarea", "bdrms", "bthrms", "floors"]
nominal_features = ["type", "devment", "ber", "location"]

# Create the preprocessor
preprocessor = ColumnTransformer(["num", StandardScaler(), numeric_features],
                                  ["nom", OneHotEncoder(handle_unknown="ignore"), nominal_features],
                                  remainder="drop")

# Run the preprocessor
preprocessor.fit(df)
X = preprocessor.transform(df)
```

In [ ]:
```python
# Create the clustering object
k = 2
kmeans = KMeans(n_clusters=k)
```
In [1]:
# Run it
kmeans.fit(X)

In [2]:
# In case you're interested, you can see the final inertia
kmeans.inertia_

In [3]:
# ...and even the vectors of the final centroids
kmeans.cluster_centers_

In [4]:
# The clusters have been labeled (numbered) from 0...(k-1)
# We can see the labels of each example in the dataset
kmeans.labels_

In [5]:
# Let's hack up a function that helps us look at a few examples from each cluster

def inspect_clusters(alg, df, k, features_to_show, how_many_to_show=None):
    for i in range(0, k):
        print("A few examples from cluster ", i)
        indexes = alg.labels_ == i
        max_available = indexes.sum()
        print(df.loc[indexes, features_to_show]
              [:max_available if not how_many_to_show else min(how_many_to_show, max_available)])
    print()

In [6]:
# Show 3 examples from each cluster for KMeans with random initialization
inspect_clusters(kmeans, df, k, numeric_features + nominal_features, 3)

- Go back and try with a different value for k
- But eye-balling examples from the clusters is not a reliable way of judging the quality of the clustering

**Evaluating clustering, part 1**

- Suppose someone has already done a manual clustering of the dataset ('ground truth'):
  - Then you can compare the output of the algorithm with the ground truth
  - Discussed in next lecture
- Suppose you don't have a ground truth (much more typical!):
  - **Silhouette Coefficient** is one of several ways of scoring clustering quality:
    - For each example \( x \in X \), compute
      \[
      \frac{b - a}{\max(a, b)}
      \]
      where \( a \) is the mean distance between \( x \) and all other examples in the same cluster, \( b \) is the mean distance between \( x \) and all examples in the next nearest cluster
    - The Silhouette Coefficient is the mean of all of these
    - Its values lies in \([-1, 1]\):
      - Positive values suggest examples are in their correct clusters
      - Values near 0 indicate clusters that are not well separated
      - Negative values suggest examples are in the wrong clusters

In [7]:
silhouette_score(X, kmeans.labels_, metric='euclidean')

In [8]: