CS4618: Artificial Intelligence I

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.pipeline import Pipeline
from sklearn.pipeline import FeatureUnion
from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette_score

# Class, for use in pipelines, to select certain columns from a DataFram
e and convert to a numpy array
# From A. Geron: Hands-On Machine Learning with Scikit-Learn & TensorFlo
w, O'Reilly, 2017
# Modified by Derek Bridge to allow for casting in the same ways as pand
as.DataFrame.astype
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names, dtype=None):
        self.attribute_names = attribute_names
        self.dtype = dtype
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X_selected = X[self.attribute_names]
        if self.dtype:
            return X_selected.astype(self.dtype).values
        return X_selected.values

# Class, for use in pipelines, to binarize nominal-valued features (whil
e avoiding the dummy variable trap)
# By Derek Bridge, 2017
class FeatureBinarizer(BaseEstimator, TransformerMixin):
    def __init__(self, features_values):
        self.features_values = features_values
        self.num_features = len(features_values)
        self.labelencodings = [LabelEncoder().fit(feature_values) for fe
ature_values in features_values]
        self.onehotencoder = OneHotEncoder(sparse=False,
            n_values=[len(feature_values) for feature_values in features
_values])
        self.last_indexes = np.cumsum([len(feature_values) - 1 for featu
re_values in self.features_values])
    def fit(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        return self.onehotencoder.fit(X)
    def transform(self, X, y=None):
        for i in range(0, self.num_features):
            X[:, i] = self.labelencodings[i].transform(X[:, i])
        onehotencoded = self.onehotencoder.transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def fit_transform(self, X, y=None):
        onehotencoded = self.fit(X).transform(X)
        return np.delete(onehotencoded, self.last_indexes, axis=1)
    def get_params(self, deep=True):
        return {"features_values" : self.features_values}
    def set_params(self, **parameters):
        for parameter, value in parameters.items():
            self.setattr(parameter, value)
        return self
```

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) $\boldsymbol{X}$ it returns a *partition* of $\boldsymbol{X}$ 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

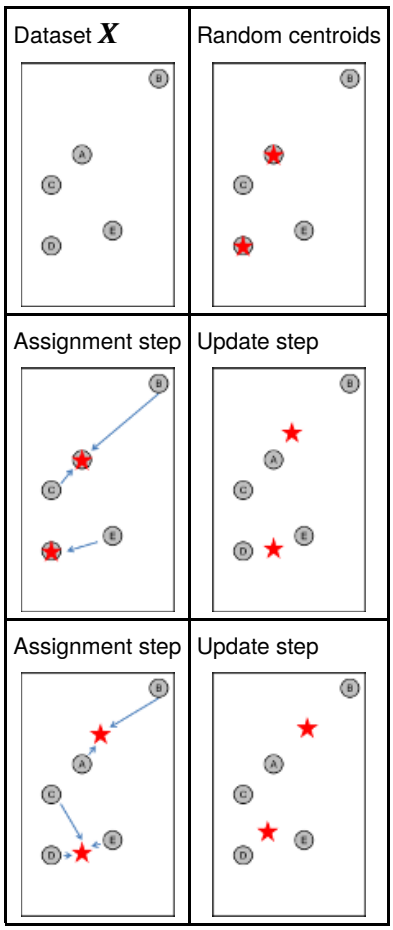
Centroids

- Class exercise: What are the centroids of these clusters?
  1. $\{\begin{bmatrix}1\\1\\1\end{bmatrix}, \begin{bmatrix}2\\4\\6\end{bmatrix}, \begin{bmatrix}3\\7\\11\end{bmatrix}\}$
  2. $\{\begin{bmatrix}4\\3\\7\end{bmatrix}\}$
  3. $\{\begin{bmatrix}2\\3\end{bmatrix}, \begin{bmatrix}4\\2\end{bmatrix}\}$
- 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 $\boldsymbol{X}$ to be the initial centroids, e.g. randomly
- Then, repeatedly,
  - **Assignment step**: Each example $\boldsymbol{x} \in \boldsymbol{X}$ is assigned to one of the clusters: the one whose centroid is closest to $\boldsymbol{x}$
  - **Update step**: It re-computes the centroids of the clusters

Toy Example, $k = 2$



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 equaly 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

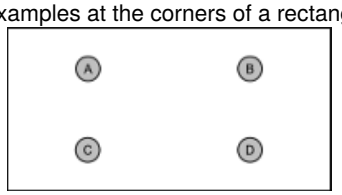
- What is k-means trying to achieve?
  - A clustering that **minimizes inertia**: the within-cluster sum of distances
  - I.e. the sum of the distances from each $\boldsymbol{x} \in \boldsymbol{X}$ to its centroid is as low as possible

  (Advanced: The algorithm is more correctly formalized as trying to minimise the within-cluster sum of squares of disstances, 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, $\boldsymbol{X}$ contains four examples at the 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 $\boldsymbol{X}$ has problems:
  - The algorithm can return different clusters for $\boldsymbol{X}$ 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 $Centroids$ an example $\boldsymbol{x} \in \boldsymbol{X}$, chosen at random with uniform probability
  - while $|Centroids| < k$
    - insert into $Centroids$ a different example $\boldsymbol{x} \in \boldsymbol{X}$, chosen with probability proportional to $(\min_{\boldsymbol{x}' \in Centroids} \text{dist}(\boldsymbol{x}, \boldsymbol{x}'))^2$
- 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 `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 include dimensionality reduction such as PCA in your pipeline
- The algorithm mostly scales well to larger data
  - There are variants for speed-up, e.g. `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 [ ]:
```
# Use pandas to read the CSV file into a DataFrame
df = pd.read_csv("datasets/dataset_corkA.csv")
```

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

# Create the pipelines
numeric_pipeline = Pipeline([
        ("selector", DataFrameSelector(numeric_features)),
        ("scaler", StandardScaler())
    ])

nominal_pipeline = Pipeline([
        ("selector", DataFrameSelector(nominal_features)),
        ("binarizer", FeatureBinarizer([df[feature].unique() for feature
in nominal_features]))])

pipeline = Pipeline([("union", FeatureUnion([("numeric_pipeline", numeri
c_pipeline),
                                             ("nominal_pipeline", nomina
l_pipeline)]))])
```

In [ ]:
```
# Run the pipeline
pipeline.fit(df)
X = pipeline.transform(df)
```

In [ ]:
```
# Create the clustering object

k = 2
kmeans = KMeans(n_clusters=k)
```

In [ ]:
```
# Run it
kmeans.fit(X)
```

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

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

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

In [ ]:
```
# Let's hack up a function that helps us look at a few examples from eac
h 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.ix[indexes, features_to_show]
                   [:max_available if not how_many_to_show else min(how_
many_to_show, max_available)])
        print()
```

In [ ]:
```
# Show 3 examples from each cluster for KMeans with random initializatio
n
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 $\boldsymbol{x} \in \boldsymbol{X}$, compute

      $$\frac{b - a}{max(a, b)}$$

      where a is the mean distance between $\boldsymbol{x}$ and all other examples in the same cluster, b is the mean distance between $\boldsymbol{x}$ 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 [ ]:
```
silhouette_score(X, kmeans.labels_, metric='euclidean')
```