Lecture Five: Unsupervised learning - Clustering

COMP3032 Machine Learning

Unsupervised learning

- vast majority of the available data is unlabeled: we have the input features X, but we do not have the labels y
- unsupervised learning
- clustering is usually the first choice when need to add meaning to unlabeled data

Clustering

The task of identifying similar instances and assigning them to clusters (groups).

- clustering is an algorithm for unsupervised learning task
- the goal is to group similar instances together into clusters
- minimizes the intracluster distances and maximizes the intercluster distances
  - samples that are in the same cluster to be as similar as possible
  - samples from different clusters to be as different as possible

Clustering

Clustering is a great tool for

- data analysis
- customer segmentation
- recommender systems
- anormaly detection
- semi-supervised learning
- search engine
- image segmentation
- dimensionality reduction

Why Clustering

- data analysis: run a clustering algorithm first, and then analyze each cluster separately
- customer segmentation: cluster customers based on their purchases and activities on the website
    - help to understand your customers and what they need
    - adapt your products and marketing campaigns to each cluster
- recommender systems: suggest content/product that other users in the same cluster like
- anormaly detection: any instance with a low affinity to all the clusters is likely to be an anomaly
    - Useful for fraud detection

Why Clustering

- semi-supervised learning: Perform clustering and propagate the labels to all the instances in the same cluster
  - greatly increase the number of labels available
  - can then apply a supervised learning algorithm
  - improve its performance
- search engines: search for images similar to a reference image
  - first apply a clustering algorithm to all the images in database
  - similar images would be in the same cluster
  - when provided with a reference image, use the trained clustering model to find this image's cluster
  - then return all the images from this cluster

Why Clustering

- image segmentation: clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster
  - possible to considerably reduce the number of different colors in the image
  - makes it easier to detect the contour of each object
  - used in many object detection and tracking systems

What is a cluster

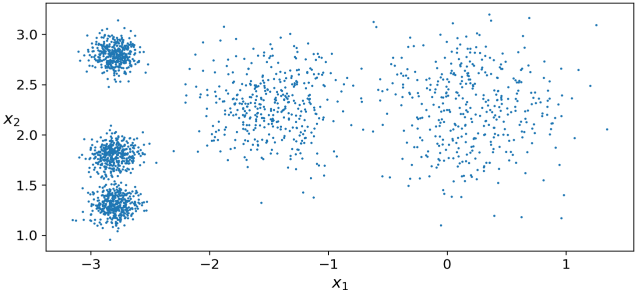
- no universal definition
- depends on the context
- different algorithms will capture different kinds of clusters
- K-Means is a popular clustering algorithm

K-Means

- proposed by Stuart Lloyd at Bell Labs in 1957, but published in 1982
- as a technique for pulse-code modulation
- In 1965 Edward W. Forgy published virtually the same algorithm
- sometimes referred to as Lloyd-Forgy

K-Means

A simple algorithm capable of clustering dataset as follows very quickly and efficiently, often in just a few iterations



An example: train and predict

- Train a K-Means clusterer on the dataset using *KMeans* from *sklearn.cluster*.
- It will try to find each blob's center and assign each instance to the closest blob
- Note that you need to specify the number of clusters k for the algorithm to find
- In general it is not that easy

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

An example: five clusters

- Each instance was assigned to one of the five clusters
- An instance's label is the index of the cluster that this instance is assigned to
- The KMeans instance preserves the labels of the instances it was trained on
- available via the instance variable *kmeans.labels_*

```
>>> y_pred
# array([0, 4, 1, ..., 3, 1, 4])
>>> kmeans.labels
# array([0, 4, 1, ..., 3, 1, 4])
```

An example: five centroids

- take a look at the five centroids that the algorithm found via *kmeans.cluster_centers_*
- assign new instances to the closest cluster and predict
- the vast majority were assigned to the appropriate cluster
- a few instances were probably mislabeled

```
>>> kmeans.cluster_centers_
# array([[-2.80389616,  1.80117999],
[ 0.20876306,  2.25551336],
[-2.80037642,  1.30082566],
[-1.46679593,  2.28585348],
[-2.79290307,  2.79641063]])
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 4, 4])
```

An example: soft clustering

- assigning each instance to a single cluster is called hard clustering
- giving each instance a score per cluster is called soft clustering
- the score can be the distance between the instance and the centroid
- or can be a similarity score (or affinity)
- the *transform*() method measures the distance from each instance to every centroid
- high-dimensional dataset can be transformed this way, to a k-dimensional dataset
- the transformation can be a very efficient nonlinear dimensionality reduction technique

An example: the transform

```
>>> kmeans.transform(X_new)
# array([[2.81093633, 0.32995317, 2.88633901, 1.49439034,
                                              2.9042344 ],
[5.80730058, 2.80290755, 5.84236351, 4.4759332 ,
                                              5.84739223],
[1.21475352, 3.29399768, 1.71086031, 1.69136631,
                                              0.29040966],
[0.72581411, 3.21806371, 1.21567622, 1.54808703,
                                             0.36159148]])
```

The K-Means algorithm: no given centroids

1. pick K random points and set them as the cluster centroids
2. assigns each data point to the nearest centroid to form K clusters
3. updates a centroid
4. go back to step 2 to reassign the samples based on the updated centroids
5. if the centroids didn't move much, stop. (the algorithm has converged)

The K-Means algorithm: no given centroids

- this is an iterative algorithm
- it keeps iterating until it converges
- can limit the number of iterations by setting its *max_iter* hyperparameter
- it is guaranteed to converge in a finite number of steps (usually quite small)
- it may converge to a local optimum, depends on the centroid initialization

Centroid initialization methods: if know the centroids

- sometimes happen to know approximately where the centroids should be
- e.g. from another clustering algorithm
- can set the init hyperparameter to a NumPy array containing the list of centroids, and set *n_init* to 1

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1],
                                        [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1
```

The K-Means algorithm: run multiple times

- different initial cluster centroids may lead to different results
- can run the algorithm multiple times with different random initializations and keep the best solution
- The number of random initializations is controlled by the *n_init* hyperparameter
- by default, it is equal to 10,
- the whole algorithm runs 10 times when you call fit()
- Scikit-Learn keeps the best solution

The K-Means algorithm: performance

- how to know which solution is the best?
- It uses a performance metric, called the model's inertia
- it is the mean squared distance between each instance and its closest centroid
- The KMeans class runs the algorithm *n_init* times and keeps the model with the lowest inertia
- a model's inertia is accessible via the *inertia_* instance variable
- The score() method returns the negative inertia

```
>>> kmeans.inertia_
211.59853725816856
>>> kmeans.score(X)
-211.59853725816856
```

The K-Means++ algorithm

- an important improvement to the K-Means algorithm
- has a smarter initialization step that tends to select centroids that are distant from one another
- this makes the K-Means algorithm much less likely to converge to a local optimal
- possible to greatly reduce the interation times to find the optimal solution
- the KMeans class uses this initialization method by default
- involve additional computation

Accelerated K-Means

- proposed in 2003 by Charles Elkan
- Considerably accelerates by avoiding many unnecessary distance calculations
- exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points)
- keep track of lower and upper bounds for distances between instances and centroids
- the KMeans class uses by default

Mini-batch K-Means

- proposed in 2010 by David Sculley
- instead of using the full dataset at each iteration, it uses mini-batches
- move the centroids just slightly at each iteration
- can speed up the algorithm typically by a factor of three or four
- make it possible to cluster huge datasets that do not fit in memory
- its inertia is generally slightly worse, especially as the number of clusters increases

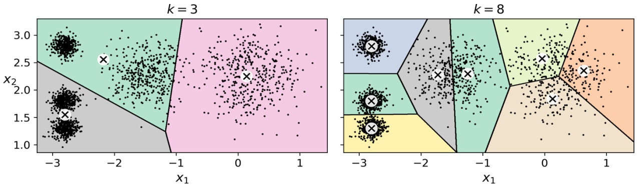
Mini-batch K-Means

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

Finding the optimal number of clusters

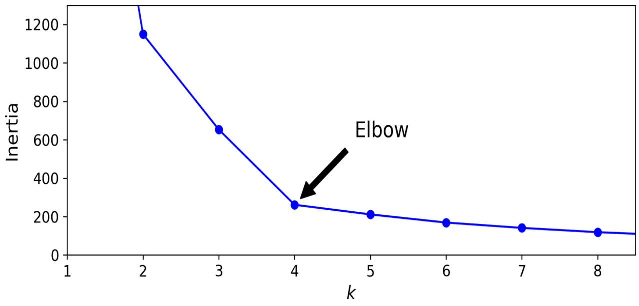
- in general, not so easy to know how to set k
- the result might be quite bad if you set a wrong value



Plot the inertia as a function of k

- minimizing inertia is not a good performance metric for choosing k because it keeps getting lower as k increases
- plot the inertia as a function of k
- the inertia drops very quickly as k increases up to 4
- then it decreases much more slowly when keep increasing k
- the curve has roughly the shape of an arm, and there is an elbow at k $=$ 4
- 4 would be a good choice if not know better
- any lower value would be dramatic, while any higher value would not help much

Plot the inertia as a function of k



Silhouette score

- a more precise approach is to use the silhouette score
- more computationally expensive
- a measure of how similar a sample is to its own cluster compared to the samples in the other clusters

Silhouette score

- a is the mean distance of an instance to the other instances in the same cluster (the mean intra-cluster distance)
- b is the mean distance between the same instance and all the other instances in the nearest cluster (the mean nesrest-cluster distance)
  - defined as the one that minimizes b, excluding the instance's own cluster
- an instance's silhouette coefficient is equal to $(b - a)/max(a, b)$
- silhouette score is the mean silhouette coefficient over all the instances

Silhouette coefficient

- The silhouette coefficient can vary between -1 and $+1$
- A coefficient close to $+1$ means that the instance is well inside its own cluster and far from other clusters
- a coefficient close to 0 means that it is close to a cluster boundary
- a coefficient close to -1 means that the instance may have been assigned to the wrong cluster

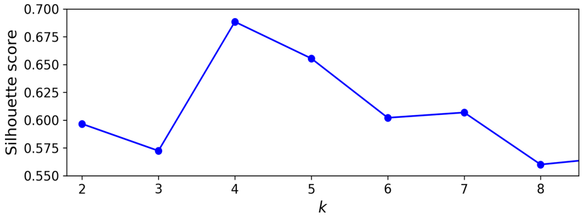
Silhouette score

- To compute the silhouette score, use Scikit-Learn's *silhouette_score*() function
- Use all the instances in the dataset and the labels they were assigned

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

Compare silhouette scores

- compare the silhouette scores for different numbers of clusters
- k=4 is a very good choice, k=5 is good as well
- much better than k=2,3, 6, 7 or 8
- the visualization is much richer than the previous one



Using Clustering for Preprocessing

- an efficient preprocessing step before a supervised learning algorithm
- load the digits dataset, a simple MNIST-like dataset containing 1,797 grayscale 8 x 8 images representing the digits 0 to 9
- fit the logistic regression model directly, evaluate the accuracy
- using K-Means as a preprocessing step, then apply a Logistic Regression model, evaluate the accuracy
- reduced the error rate by almost 30% (from about 3.1% to about 2.2%)

The code

```
#load data
from sklearn.datasets import load_digits
X_digits, y_digits = load_digits(
return_X_y=True)
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(
X_digits, y_digits, random_state=42)
```

The code

```
#fit a Logistic Regression model and evaluate
#it on the test set:
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(
multi_class="ovr", solver="lbfgs",
max_iter=5000, random_state=42)
```

The code

```
log_reg.fit(X_train, y_train)
log_reg_score = log_reg.score(X_test, y_test)
print("logistic-reg score= ", log_reg_score)
#logistic-reg score=  0.9688888888888889
```

The code

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([
("kmeans", KMeans(n_clusters=50, random_state=42)),
("log_reg", LogisticRegression(multi_class="ovr",
solver="lbfgs", max_iter=5000, random_state=42)),
])
pipeline.fit(X_train, y_train)
```

The code

```
pipeline_score = pipeline.score(X_test, y_test)
print("pineline score=", pipeline_score)
#pineline score= 0.9777777777777777
```

Spectral clustering

- instead of clustering in the original space, first map data to a new sapace with reduced dimensinality
- similarities are made more apparent in there
- any feature selection or extraction method can be used for this purpose
- then cluster in the new space

Spectral clustering

- from sklearn.cluster import SpectralClustering
- take a similarity matrix between the instances and creates a low-dimensional embedding from it
- i.e. reduces dimensionality
- then use another clustering algorithm in this low-dimensional space
- Scikit-Learn's implementation uses K-Means

Spectral clustering

- can capture complex cluster structures
- can be used to cut graphs
- identify clusters of friends on a social network
- does not scale well to large numbers of instances
- it does not behave well when the clusters have very different sizes