data_clustering_1

June 27, 2018

1 Data Clustering

Data clustering is a process of assigning a set of records into subsets, called clusters, such that records in the same cluster are similar and records in different clusters are quite distinct.

A typical clustering process involves the following five steps:

1. pattern representation;
2. dissimilarity measure definition;
3. clustering;
4. data abstraction;
5. assesment of output

In this interactive session, we will be reviewing the *k-means* algorithm.

1.1 The *k-means* Algorithm

The *k-means* algorithm is the most popular and the simplest partitional clustering algorithm. It has many variations. This exercise will review the standard algorithm and several implementations (possibly for different variations).

1.1.1 Description of the Algorithm

Let $X = \{\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1}\}$ be a numeric dataset containing n records and k be an integer in $\{1, 2, \cdots, n\}$. The *k-means* algorithm tries to divide the dataset into k clusters C_0, C_1, $\cdots$, and C_{k-1} by minimizing the following objective function:

$$E = \sum_{i=0}^{k-1} \sum_{\mathbf{x} \in C_i} D(\mathbf{x}, \boldsymbol{\mu}_i),$$

where $D(\cdot, \cdot)$ is a distance measure and $\boldsymbol{\mu}_i$ is the mean of cluster C_i, i.e.,

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$$

Let γ_i be the cluster membership of record $\mathbf{x}_i$ for $i = 0, 1, \cdots, n-1$. That is, $\gamma_i = j$ if $\mathbf{x}_i$ belongs to cluster C_j. Then the objective function can be rewritten as:

$$E = \sum_{i=0}^{n-1} D(\mathbf{x}_i, \boldsymbol{\mu}_{\gamma_i})$$

To minimize the objective function, the *k-means* algorithm employs an iterative process. At the beginning, the *k-means* algorithm selects k random records from the dataset X as initial cluster centers.

Suppose $\boldsymbol{\mu}_0^{(0)}$, $\boldsymbol{\mu}_1^{(0)}$, $\cdots$, and $\boldsymbol{\mu}_{k-1}^{(0)}$ are the initial cluster centers. Based on these cluster centers, the *k-means* algorithm updates the cluster memberships $\gamma_0^{(0)}$, $\gamma_1^{(0)}$, $\cdots$, $\gamma_{n-1}^{(0)}$ as follows:

$$\gamma_i^{(0)} = \underset{0 \le j \le k-1}{\operatorname{argmin}} D(\mathbf{x}_i, \boldsymbol{\mu}_j^{(0)}) \tag{1}$$

where argmin is the argument that minimizes the distance. That is, $\gamma_i^{(0)}$ is set to the index of the cluster to which $\mathbf{x}_i$ has the smallest distance.

Based on the cluster memberships $\gamma_0^{(0)}$, $\gamma_1^{(0)}$, $\cdots$, $\gamma_{n-1}^{(0)}$, the *k-means* algorithm updates the cluster centers as follows:

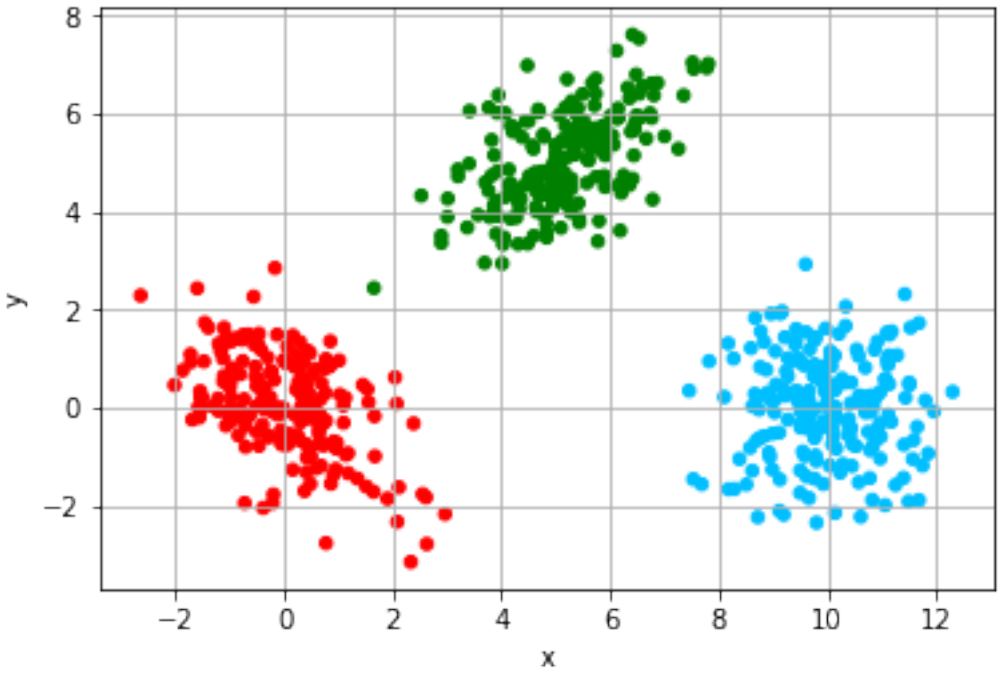
$$\boldsymbol{\mu}_j^{(1)} = \frac{1}{\left|\left\{i \middle| \gamma_i^{(0)} = j\right\}\right|} \sum_{i=0, \gamma_i^{(0)}=j} \mathbf{x}_i, \qquad j = 0, 1, \cdots, k-1 \tag{2}$$

Then the *k-means* algorithm repeats updating the cluster memberships based on equations 1 and 2.

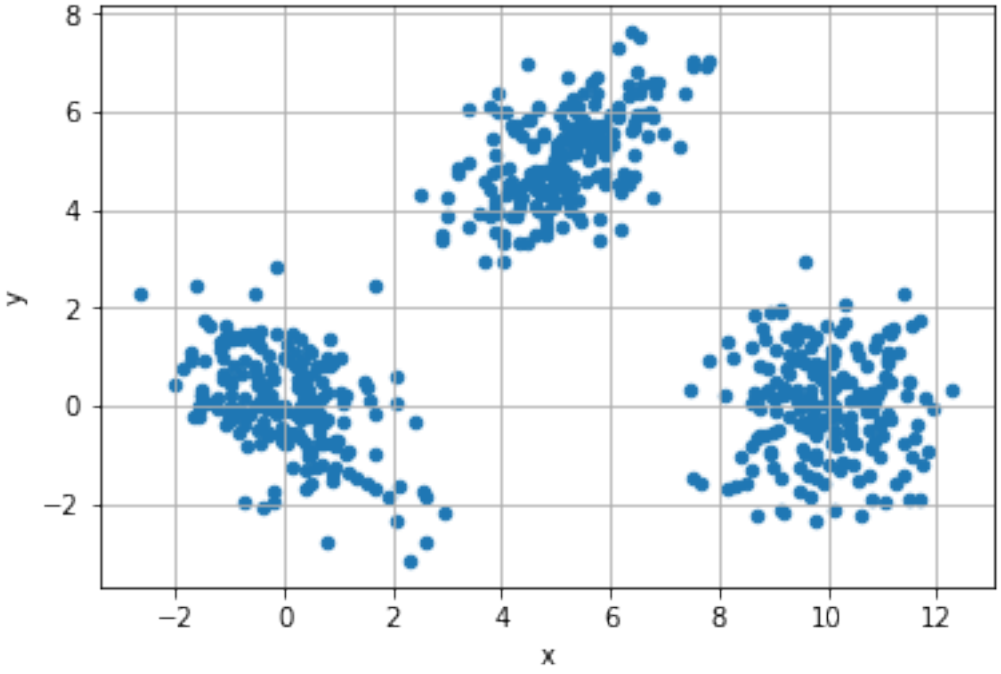
1.2 Exercises

1.2.1 Visualization of dataset for exercises

```
In [1]: %matplotlib inline

In [2]: import pandas as pd

        input_file = "data/600points.csv"

        # Input file has labeled records. 'label' is one of 1, 2 or 3.
        labeled_df = pd.read_csv(input_file,
                                 header=None,
                                 names=['id', 'x', 'y', 'label'])
        labeled_df.dtypes

Out[2]: id         int64
        x        float64
        y        float64
        label      int64
        dtype: object

In [3]: # To visualize "named" colors see:
        # https://matplotlib.org/1.4.3/examples/color/named_colors.html
        label_to_color = {1: 'red', 2: 'deepskyblue', 3: 'green'}
        colors = labeled_df.loc[:, 'label'].map(label_to_color)
        labeled_df.plot.scatter('x', 'y', grid=True, c=colors);
```



In [4]:
```
# Unlabeled data (prior to "clustering").
df = labeled_df.loc[:,['x', 'y']]
df.plot.scatter('x', 'y', grid=True);
```



1.2.2 To practice your Python

1. From the datasets card in trello, use data.tar.gz to get the file with name 600points.csv.
2. Use the kmeans implementation from scikit-learn to run the algorithm for k = 3 clusters and a max of 100 iterations. You can install that using pip (`$ pip install scikit-learn`) or anaconda, (`$ conda install scikit-learn`).
3. Use python's docopt or `argparse` (available in the standard library) to parse the following command line options:

```
Allowed options:
  --help                produce help message
  --datafile arg        the data file
  --k arg (=3)          number of clusters
  --maxiter arg (=100)  maximum number of iterations
```

4. Implement the algorithm by yourself and compare with the results obtained with scikit-learn.
5. Use numba to try to improve the performance of your implementation.
6. Use matplotlib (or another library of your preference) to visualize data and your results. Fine tune your plot so that the assesment of the result is easier to understand.

1.2.3 To practice with .NET (C# or F#)

1. Use the *k-means* implementation from version 0.2 of ML.NET as a reference implementation for comparison (correctness of results, performance, etc).
2. Use .NET's docopt port to parse command line options for your program (see analogous section for Python).
3. Implement your own version of k-means using your preferred .NET language (C#, F#, VB.NET).
4. Visualize your results with XPlot, FSharp.Charting, Live-Charts or any other good quality library of your preference.

1.2.4 To practice with GPUs and Python/R/C

1. Take a look at kmcuda. See what the possibilities are for your favorite programming language.
2. Visualize your results with the best library for your language of choice.

1.2.5 To practice with GPUs or Multicore CPUs and Haskell

1. Take a look at Accelerate and the different supported backends (e.g. Multicore CPUs and GPUs).
2. Study and explain the *k-means* sample
3. Compare the implementations and performance.
4. Visualize using `matplotlib` bindings for Haskell.

1.2.6 To practice with Scala and Spark

1. Try to use the existing Scala implementations:

   - With Spark's MLLib
   - With Spire

2. Study and explain.

3. Hard: study if it is possible to improve performance using sbt-javacpp bindings for CUDA and previous implementation code/algorithms.

4. Visualize using Vegas or any other good quality library of your preference.