Софиийски университет "Св. Климент Охридски"
Факултет по математика и информатика

Домашна работа 2

по

Изкуствен Интелект

спец. Информатика, 3 курс, летен семестър,

учебна година 2018/19

Тема: Клъстеризация с помощта на метода k-Means

2 юни 2019 г.

София

Изготвил:

Иво Алексеев Стратев

Фак. номер: 45342

Група: 3

Съдържание

1 Задача

Входните данни са добре известното множество "Iris". Първоначално това множество е публикувано в UCI Machine Learning Repository: Iris Data Set. Всеки ред на таблицата описва цвете ирис, представено с размерите на неговите ботанически части в сантиметри. Таблицата включва описания на 150 примера в термините на 4 атрибута. Представящи ботаническите параметри на цветето. (http://archive.ics.uci.edu/ml/datasets/Iris).
Задачата е да се имплементира метода k-Means за групиране на примерите от това множество в три клъстера. Използвайки атрибутите: sepal length, sepal width, petal length и petal width. Множеството съдържа и класови атрибут, които групира примерите в три класа: Iris Setosa, Iris Versicolour и Iris Virginica. След, което да сравнят примерите във всеки от получените клъстери с вече известните класове. Да се използват поне две различни функции за определяне на разстоянието и да определете коя е по-точната за това множество от данни.

2 Описание на използвания метод за решаване на задачата

Имплементацията на клъстеризиращия алгоритъм представлява оптимизирана версия на алгоритъма k-Means. Оптимизациите са две:

- използва се детерминистична инициализираща фаза, вместо типичните рандомизирани инициализации с цел подобрябване на сходимостта на алгоритъма.
- стъпката е паралелна, като точките не се разделят, смятат се само новите центроиди. Точките се разделят само в завършителната фаза на реализацията.

2.1 Класическа версия на алгоритъма k-Means

В алгоритъма k е броят на клъстерите, на които искаме да разделим входните данни. Центроид наричаме центъра на всеки клъстер. Той може да бъде намерен чрез използването на различни статистически или псевдостатистически функции. За алгоритъма това е средната точка на клъстера. Тоест математическата функция е *mean*. Понеже алгоритъма е многомерен, то функцията е покомпонентна. Тоест стойността във всяка компонента е средната стойност на съответната компонента на точките от клъстера.

$$Mean(C) = \frac{1}{|C|}\sum_{p\in C} p$$

2.1.1 Инициализираща фаза

Цели да инициализира центроидите. Обикновенно се ползват рандомизирани стратегии. Две от най-известните стратегии са:

- На произволен принцип се избират k точки от множеството.
- На всяка точка по произволен начин се съпоставя клъстер, след което се смятат центроидите на всеки от получените клъстери.

2.1.2 Стъпка

Докато не настъпи определено събитие се повтаря следната логика: Взимат се k празни множества, представляващи всеки клъстер. Всяко множество съотвества на точно един центроид. За всяка точка се пресмята разтоянието да всеки центроид. Точката се премества в съответния клъстер (множество). Пресмятат се медианите на всеки клъстер (множество). Това са новите центроиди. Събитията обикновенно са:

- достигнат е определен брой итерации
- Във всеки клъстер по-малко от определен брой точки са били преместени
- Центроидите са се преместили на по-малко от определено разтояние

Следва подобронно описание на реализацията.

2.2 Инициализираща фаза

Тъйкато рандомизираните методи не гарантират по никакъв начин сходимостта на алгоритъма. То е използван следния подход за инициализация, който е със сложност $\Theta(n)$, където n е броя на точките. Тоест сложността е същата като съпоставянето на произволен начин на клъстер. Вземат се първите k точки от множеството. Това са началните центроиди. След, което за всяка точка се пресмята разтоянието до всеки клъстер. Ако разтоянието между точката и някой клъстер е по-малко от разтоянието от някой центроид до друг, то се обновява някой центроид на най-късо разтояние до точката по формулата:

$$\left(\frac{|C_i|}{|C_i|+1}.centroid(C_i)\right)+\left(\frac{1}{|C_i|+1}.point\right) \tag{1}$$

където C_i е избрания клъстер. Ако пък разтоянието от точката до всеки клъстер е по-голямо от разтоянията от всеки клъстер до всеки друг, то се избират два клъстера с минимално разтояние между тях, които да бъдат обединени. Като в резултат центроидите на двата клъстера се заменят с нов центроид пресмет по формулата:

$$\left(\frac{|C_i|}{|C_i|+|C_j|}.centroid(C_i)\right)+\left(\frac{|C_j|}{|C_i|+|C_j|}.centroid(C_j)\right) \tag{2}$$

и се добавя нов клъстер с добавянето на нов центроид, който е разглежданата точка.

2.3 Стъпка

Лесно се забелязва, че в стъпката на оргиналния вариант на k-Means точките биват разделяни единствено за да могат да бъдат пресметнати новите центроиди. Но това е излишно понеже новите центроиди могат да бъдат пресметнати в движение. При това пресмятането на новите центроиди може да бъде извършено паралелно. За това стъпката в предложената реализация е следната: Множеството от точки се разделя на H части, за всяка част паралелно се пресмятат временните нови центроиди за съответната част. Тоест за всяка точка във фиксирана част се извършват следните стъпки. Намира се клъстера с най-късо разтояние на центроида си до точката, с най-малък брой точки в себе си. След което ако клъстера е празен, то временния нов центроид е точката. В противен случай се използва формула 1 за обновяване на временния центроид. След като за всяка част бъдат намерени временните нови центроиди се обединяват по формулата:

$$\left(\frac{|temp_i|}{|current_i|+|temp_i|}.centroid(temp_i)\right)+\left(\frac{|current_i|}{|current_i|+|temp_i|}.centroid(current_i)\right) \tag{3}$$

Където $temp_i$ съотвества временния клъстер резултат от обединението на центроидите или празното множество. $current_i$ е клъстера съотвестваш на i-тия центроид от някоя част. Пълната формула, по която се пресмятат новия i-ти центроид съотвестваш на текущия i-ти центроид е:

$$\sum_{j=1}^{H}\frac{|temp_{ji}|}{n}.centroid(temp_{ji}) \tag{4}$$

Където $temp_{ji}$ съотвества на получения центроид съотвестваш на i-тия досегашен центроид от j-тата част.

2.4 Финална фаза

Точките се разпределят в k клъстера спрямо това до кой центроид се намират най-близо. В реализацията клъстер представлява масив от индекси. Всеки индекс е идексът на съответната точка от входния масив с точки.

3 описание на реализацията с псевдокод

Ще използваме следните константи:

- K - брой клъстери (центроиди)
- N - размерност на точките (брой координати)
- I - максимален брой итерации
- H - брой нишки (части, на които да се разделят входните данни)
- e - достатъчно малко число избрано от потребителя

```
function K_MEANS(points)
    means ← INITIALMEANS(points)
    for i ← 1 to I do
        newMeans ← CENTROIDS(points, means)
        if ARECLOSE(newMeans, means) then break
        end if
        means ← newMeans
    end for
    return PARTITION(points, means)
end function
```

```
function INITIALMEANS(points)
    means ← FIRSTASMEANS(points)
    dists ← MEANSDISTS(means)
    minDist ← MINMEANSDIST(dists)
    count ← points.size
    for i ← k + 1 to count do
        point ← points[i]
        distsFromPointToMeans ← DISTSTOMEANS(point, means)
        minDistPoint ← MINDISTTOMEANS(distsFromPointToMeans)
        if minDistPoint.dist < distOfMinDist.dist then
            UPDATEMEANSWITHPOINT(means, dists, minDist, point, minDistPoint.toCentroid)
        else
            UPDATEMEANSWITHMERGE(means, dists, minDist, point, distsFromPointToMeans)
        end if
    end for
    return means
end function
```

```
function CENTROIDS(points, means)
    partMeans ← PORTIONMEANS(points, means)
    return NEWMEANSFROMPORTIONMEANS(partMeans)
end function
```

```
function ARECLOSE(means1, means2)
    for all mean : means1 do
        if not HASCLOSE(mean, means) then
            return false
        end if
    end for
    return true
end function
```

```
function PARTITION(points, means)
    clusters[K]
    for i ← 1 to K do
        clusters[i].centroid ← means[i].centroid
    end for
    count ← points.size
    for index ← 1 to count do
        point ← points[index]
        centroidIndex ← 1
        d ← DIST(point, means[1].centroid)
        for j ← 2 to K do
            dj ← DIST(point, means[j].centroid)
            if dj < d ∨ dj = d ∧ clusters[j].size < clusters[centroidIndex].size then
                centroidIndex ← j
                d ← dj
            end if
        end for
        PUSH(clusters[centroidIndex].indeces, index)
    end for
    return clusters
end function
```

```
function FIRSTASMEANS(points)
    means[K]
    for i ← 1 to K do
        means[K] ← POINTMEAN(points[i])
    end for
    return means
end function
```

```
function meansDists(means)
    dists[K][K]
    for i ← 1 to K do
        for j ← 1 to i − 1 do
            d ← dist(means[i].centroid, means[j].centroid)
            dists[i][j] ← d
            dists[j][i] ← d
        end for
        dists[i][i] ← 0
    end for
    return dists
end function
```

```
function minMeansDist(dists)
    from ← 1
    to ← 2
    d ← dists[from][to]
    for i ← 1 to K do
        for j ← 1 to i − 1 do
            if dists[i][j] < d then
                from ← i
                to ← j
                d ← dists[from][to]
            end if
        end for
    end for
    return < from, to, d >
end function
```

```
function distsToMeans(point, means)
    dists[K]
    for i ← 1 to K do
        dists[i] ← dist(point, means[i].centroid)
    end for
    return dists
end function
```

```
function minDistToMeans(dists)
    to ← 1
    d ← dists[to]
    for i ← 2 to K do
        if dists[i] < d then
            to ← i
            d ← dists[to]
        end if
    end for
    return < to, d >
end function
```

```
function updateMeansWithPoint(means, dists, minDist, point, centroidIndex)
    means[centroidIndex] ← updateMean(means[centroidIndex], point)
    for i ← 1 to K do
        if i ≠ centroidIndex then
            d ← dist(means[centroidIndex].centroid, means[i].centroid)
            dists[centroidIndex][i] ← d
            dists[i][centroidIndex] ← d
            if d < minDist.dist then
                minDist ←< centroidIndex, i, d >
            end if
        end if
    end for
end function
```

```
function updateMeansWithMerge(means, dists, minDist, point, distsToMeans)
    means[minDist.from] ← mergeMeans(means[minDist.from], means[minDist.to])
    means[minDist.to] ← pointMean(point)
    for i ← 1 to K do
        if i ≠ minDist.from then
            d ← dist(means[minDist.from].centroid, point)
            dists[minDist.from][i] ← d
            dists[i][minDist.from] ← d
        end if
        if i ≠ minDist.to ∧ (i ≠ minDist.from) then
            dists[minDist.to][i] ← distsToMeans[i]
            dists[i][minDist.to] ← dists[i][minDist.to]
        end if
    end for
    minDist ← minDistToMeans(dists)
end function
```

```
function portionMeans(points, means)
    portion ← points.size / H
    futers[H]
    for i ← 1 to H do
        start ← (i − 1) ∗ portion + 1
        end ← start + portion
        if i = H then
            end ← points.size
        end if
        futers[i] ← async(task, points, means, start, end)
    end for
    return futers
end function
```

```
function newMeansFromPortionMeans(futers)
    means ← asyncResult(futers[1])
    for h ← 2 to H do
        tempMeans ← asyncResult(futers[h])
        for i ← 1 to K do
            if means[i].size = 0 ∧ tempMeans[i].size ≠ 0 then
                means[i] ← tempMeans[i]
            else
                if means[i].size ≠ 0 ∧ tempMeans[i].size ≠ 0 then
                    means[i] ← mergeMeans(means[i], tempMeans[i])
                end if
            end if
        end for
    end for
    return means
end function
```

```
function task(points, means, start, end)
    tempMeans[K]
    for i ← 1 to K do
        tempMeans[i].size ← 0
    end for
    for i ← start to end do
        point ← points[i]
        centroidIndex ← 1
        d ← dist(point, means[centroidIndex].centroid)
        for j ← 2 to K do
            dj ← dist(point, means[j].centroid)
            if dj < d ∨ dj = d ∧ tempMeans[j].size < tempMeans[centroidIndex].size then
                centroidIndex ← j
                d ← dj
            end if
        end for
        if tempMeans[centroidIndex].size = 0 then
            tempMeans[centroidIndex] ← pointMean(point)
        else
            tempMeans[centroidIndex] ← updateMean(tempMeans[centroidIndex], point)
        end if
    end for
    return tempMeans
end function
```

```
function hasClose(mean, means)
    for i ← 1 to K do
        if dist(mean.centroid, means[i].centroid) < e then
            return true
        end if
    end for
    return false
end function
```

function POINTMEAN(point)
    return $< point, 1 >$
end function

function UPDATEMEAN(mean, point)
    $tempMean \leftarrow$ POINTMEAN($point$)
    return MERGEMEANS(mean, tempMean)
end function

function MERGEMEANS(m1, m2)
    $count \leftarrow m1.size + m2.size$
    $a \leftarrow \frac{m1.size}{count}$
    $b \leftarrow \frac{m2.size}{count}$
    m
    for $i \leftarrow 1$ **to** K **do**
        $m[i] \leftarrow a.m1.centroid[i] + b.m2.centroid[i]$
    end for
    return $< m, count >$
end function

4 Инструкции за компилиране на програмата

На Linux базирана операционна система се нуждаете от **g++**, поне **5.4.0** версия. За компилиране се преместете в папката на сорс кода и от там изпълнете командата **make**. Тази команда ще стартира компилирането на кода и ще създаде изходен файл **a.out**. За да изпълните изходния файл изпълнете командата **./a.out**.

5 Примерни резултати

Използвани са 6 различни функции за определяне на разтоянието.

- L1 дистанция известна още като Манхатанско разтояние.

$$l1(x, y) = \sum_{i=1}^{d} |x_i - y_i|$$

- L2 дистанция известна още като Евклидово разтояние.

$$l2(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$$

- L3 дистанция.

$$l3(x, y) = \left(\sum_{i=1}^{d} |x_i - y_i|^3 \right)^{\frac{1}{3}}$$

- L4 дистанция.

$$l4(x,y) = \left(\sum_{i=1}^{d} |x_i - y_i|^4\right)^{\frac{1}{4}}$$

- L_∞ дистанция.

$$l_\infty(x,y) = \max\{|x_i - y_i| \mid i = 1, \dots, d\}$$

- S дистанция.

$$s(x,y) = \sum_{i=1}^{d} (x_i - y_i)^2$$

За всеки от получените клъстери се определя класа, който доминира в клъстера и се пресмята какво е отношението на точките от този клас към токчите в клъстера. За всяка дистанция се пресмята и оценяща функция на чистота на клъстеризацията. Наричана *purity* по следната формула:

$$purity(\{C_1, C_2, C_3\}) = \frac{1}{n}\sum_{i=1}^{3} \max\{|C_i \cap Class_l| \mid l \in \{Setosa, Versicolour, Virginica\}\}$$

Тази функция дава оценка за качеството на клъстеризация. За всяка функция на разтояние се извежда името на фунцкия. След, което за всеки клъстер се намира броя на точките от доминиращия клас към размера на клъстера и се извежда името на доминиращия клас. Накрая се извежда стойността на *purity* функцията за това клъстеризиране.

5.1 Изходни резултати:

L1
Virginica: 0.0540541
Versicolor: 0.234375
Setosa: 0
0.887417
L2
Versicolor: 0.222222
Setosa: 0
Virginica: 0.0526316
0.89404
L3
Versicolor: 0.222222
Setosa: 0
Virginica: 0.0526316
0.89404
L4
Setosa: 0
Virginica: 0.494949
Setosa: 0.0454545
0.662252
L_infinity
Versicolor: 0.5
Setosa: 0
Setosa: 0.0232558

0.662252
S
Versicolor: 0.222222
Setosa: 0
Virginica: 0.0526316
0.89404