

Mustererkennung: Übungsblatt 2

von

Naja v. Schmude (4127652), Lisa Dohrmann (4130066), Adrian Neumann (4140810)

Aufgabe 1

Code

Für die Implementierung des LBG Clustering Verfahrens benutzen wir im Wesentlichen zwei Funktionen. `lbg` und `getNewMeans` aus der Klasse `Cluster`. Wir benutzen eine eigene Vektorklasse, die von `Vector` erbt und unter anderem noch eine Beschriftung der Vektoren ermöglicht. Als Abbruchsbedingung haben wir die Bewegung der Repräsentanten genommen. Es wird immer der euklidische Abstand zur letzten Position der Repräsentanten ausgerechnet, wenn er für alle Repräsentanten gleich null ist, also keine Veränderung mehr stattgefunden hat, brechen wir ab.

Wenn einer der Repräsentanten überhaupt keine Daten abbekommt, wird nochmal von vorne angefangen. Hier terminieren wir potentiell nie (oder nicht schnell genug, um einen OutOfMemory Error zu verhindern), aber das ist sehr unwahrscheinlich.

```

1  public static Set<Vektor<Float>> lbg(Set<Vektor<Float>> s, int k) {
2      if (s.isEmpty())
3          throw new RuntimeException("empty_set!");
4      Set<Vektor<Float>> reps = new HashSet<Vektor<Float>>();
5      int d = s.iterator().next().size(); // dimensionen
6      // zufällige repräsentanten finden
7      for (int i = 0; i < k; ++i) {
8          Vektor<Float> v = randomVektor(d);
9          v.label = new Integer(i);
10         reps.add(v);
11     }
12     Vektor<Float>[] oldReps = new Vektor[reps.size()];
13     for (int i = 0; i < oldReps.length; ++i)
14         oldReps[i] = Vektor.nullVektor(d);
15     float diff = 42;
16     int count = 0;
17     // solange die Differenz der alten und neuen Position aller
18     // Repräsentanten > 0 ist und wir noch nicht 100 Durchläufe haben,
19     // werden die Cluster verbessert
20     do {
21         count++;
22         diff = 0;
23         for (Vektor<Float> r : reps) {
24             diff += Vektor.distance(oldReps[r.label], r);
25             oldReps[r.label] = (Vektor<Float>) r.clone();
26         }
27         try {
28             reps = getNewMeans(s, reps); // neue Repräsentanten bestimmen
29         } catch (RuntimeException e) {
30             // Einer unserer Vektoren hat keine Daten abbekommen. Auf ein
31             // neues..
32             return Cluster.lbg(s, k);
33         }
34     } while (diff > 0 && count < 100);
35     for (Vektor<Float> r : reps) {
36         r.label = KNN.classify(r, s, 7); // Label clustern anhand der
37         // nächsten 7 Nachbarn des Repräsentanten
38     }
39     return reps;
40 }

```

In der zweiten Funktion, `getNewMeans` wird die eigentliche Arbeit getan. Mit Hilfe eines Aufrufs des KNN Algorithmus vom letzten Übungszettel klassifizieren wir alle Daten des Trainingssets und berechnen die Schwerpunkte aller Cluster, um sie dann als neue Repräsentanten zurückgeben zu können.

```

39 private static Set<Vektor<Float>> getNewMeans(Set<Vektor<Float>> daten,
40 Set<Vektor<Float>> reps) {
41     int[] labels = new int[reps.size()];
42     int d = reps.iterator().next().size(); // dimension
43     Vektor<Vektor<Float>> means = new Vektor<Vektor<Float>>();
44     // nullvektor
45     Vektor<Float> zero = Vektor.nullVektor(d);
46     // neue mittelwerte mit nullvektoren initialisieren
47     for (int i = 0; i < reps.size(); ++i)
48         means.add((Vektor<Float>) zero.clone());
49
50     // datenset partitionieren
51     int leLabel = -1;
52     for (Vektor u : daten) {
53         leLabel = KNN.classify(u, reps, 1); // zugehörigkeit zum
54         // repräsentanten bestimmen
55         ++labels[leLabel]; // die anzahl der vektoren, die ihm gehören
56         // erhöhen
57         Vektor<Float> summe = Vektor.sum(u, means.get(leLabel));
58         summe.label = leLabel;
59         means.set(leLabel, summe); // draufsummieren
60     }
61     Set<Vektor<Float>> newMeans = new HashSet<Vektor<Float>>();
62     for (int i = 0; i < reps.size(); ++i) {
63         Vektor<Float> mittelwert = null;
64         if (labels[i] == 0) {
65             // einer hat keine vektoren abbekommen
66             throw new RuntimeException("No luck...");
67         } else {
68             // den Mittelpunkt des momentanen Clusters bestimmen und als
69             // neuen Repräsentanten bestimmen
70             mittelwert = Vektor
71                 .scale(means.get(i), 1 / ((float) labels[i]));
72             mittelwert.label = means.get(i).label;
73         }
74         newMeans.add(mittelwert);
75     }
76     return newMeans;
77 }

```

Nach dem Aufruf von `lbg` haben wir also die Daten in k Cluster eingeteilt und die Repräsentanten optimal bestimmt. Um jetzt damit die Testdaten zu klassifizieren, rufen wir die Methode `test` in unsere Helpers-Klasse auf, die für alle Testdaten guckt, welche k Repräsentanten am nächsten liegen und deren Label annehmen lässt. Für uns ist k hier gleich 1, da die Anzahl der Repräsentanten mit maximal 30 sehr niedrig ist, und wir mit dem betrachten von mehreren Repräsentanten erheblich schlechtere Erkennungsraten erreichten.

```

1 public static float test(Set<Vektor<Float>> training,
2 Set<Vektor<Float>> test, int k) {
3     int sum = 0;
4     for (Vektor<Float> t : test) {
5         int l = KNN.classify(t, training, k);
6         if (t.label == l)
7             sum++;
8     }
9     return ((float) sum) / test.size();
10 }

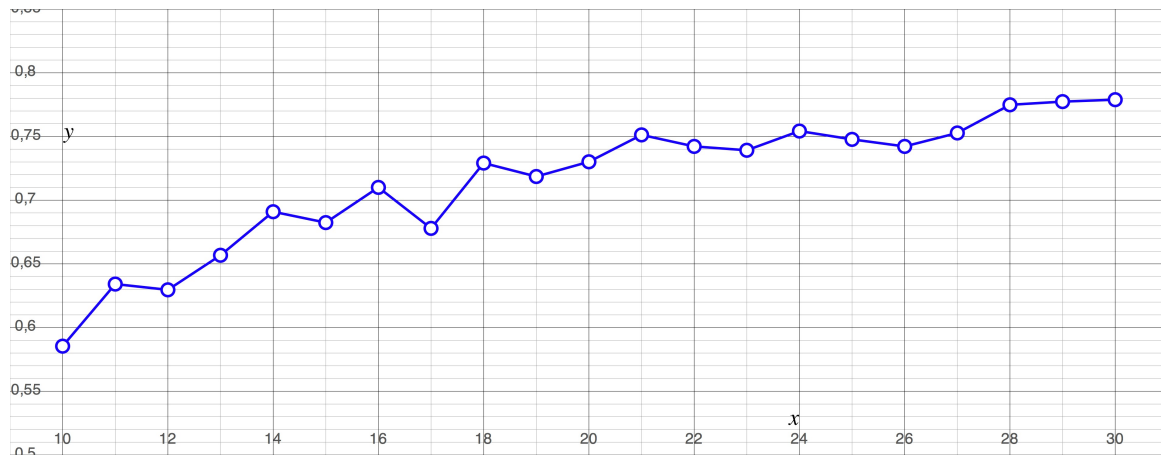
```

Auswertung

Man muss sich zuerst überlegen, was eine geeignete Definition von „gut“ ist. Offensichtlich werden die Erkennungsraten tendenziell immer besser, je mehr Repräsentanten man hat (bis man $k = n$ erreicht und wieder normales KNN macht). Außerdem ist es leicht ersichtlich, dass man mindestens so viele Repräsentanten braucht, wie man nacher Label bekommen will. Da aber die Punktwolke, die z.B. zur „7“ gehört nicht unbedingt zusammenhängend sein muss (7 kann man mit oder ohne Strich schreiben), ist es sinnvoll so viele Repräsentanten zu wählen, wie man „Ausprägungsklassen“ hat. Also zum Beispiel einen Repräsentanten für die 7 mit und einen für die 7 ohne Strich.

Wenn die Punktwolken ungefähr kugelförmig sind, sollte das ganz gut funktionieren. Sind sie eher langgestreckte Ellipsoide, fährt man womöglich besser eine Ellipse durch mehrere Repräsentanten abzudecken.

Wie man sieht ist die theoretische Überlegung, wie man k wählen sollte etwas diffus und liefert keine ordentlichen Zahlen. Deswegen haben wir uns entschieden einfach mal die Erkennungsraten für alle verlangten k durchzuprobieren. Die auftretenden Sprünge sind dabei auf das zufällige anfängliche Wählen der Repräsentanten zurückzuführen, so dass auch bei 10 Iterationen es noch zu größeren Schwankungen kommt.



Erkennungsraten für steigende Repräsentanzahlen. Gemittelt über 10 LBG Versuche