

Mustererkennung: Übungsblatt 4

von

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

Aufgabe 1

In dieser Aufgabe sollten wir den Algorithmus von Oja zum Finden der Hauptkomponenten implementieren und dann die Erkennungsraten mit den dimensional reduzierten Daten ermitteln.

Code

Der Algorithmus von Oja extrahiert in einem Schritt immer die erste Hauptkomponente. Zwischen den Schritten muss man den Einfluss dieser Komponente aus den Daten entfernen. Der Algorithmus benutzt zwei Lernfaktoren, γ und β . γ dient als Skalierungsfaktor und β ist ein Maß dafür, wie schnell γ mit den Iterationen kleiner wird.

Eine Hauptkomponente extrahieren wir mit dieser Funktion:

```

1 private static Matrix getPC(List<Digit> digits, int dim) {
2     Random r = new Random(); //zum wählen eines zufälligen Datums
3     double gammaNull = 0.2;
4     //da ich nicht richtig aufgepasst hab, als Ernesto vorgeschlagen
5     //hat, wie mit beta umzugehen sei nimmt mein gamma exponentiell
6     //ab. Deswegen hab ich mein beta viel größer gewählt, als Ernesto
7     //empfohlen hat
8     double beta = 5000;
9     //zählt Iterationen
10    int i;
11    Matrix approx=null;
12    //Wir probieren mehrere gammaNull aus, wenn wir nicht schnell
13    //genug konvergiert sind.
14    //Es scheint mit größeren gammaNull bessere Erkennungsraten zu geben,
15    //aber man konvergiert manchmal nicht.
16    do {
17        double gamma=gammaNull;
18        //zufälligen Vektor auswählen
19        approx = Matrix.random(dim,1);
20
21        //Normieren, erhöht angeblich die Stabilität
22        Matrix approxT = approx.transpose();
23        double norm = Math.sqrt(approxT.times(approx).get(0, 0));
24        approx = approx.times(1/norm);
25        Matrix approx_=new Matrix(dim,1);
26
27        i=1;
28        double dist=1; //Abstand von der letzten Iteration
29        while(dist!=0 && i < 15000) { //solange wir noch nicht konvergiert
30            sind
31
32            //Abstand zur letzten Iteration berechnen
33            Matrix tmp = approx.minus(approx_);
34            dist = tmp.transpose().times(tmp).get(0, 0);
35            //Diese Näherung merken
36            approx_=(Matrix)approx.clone();
37
38            //Mit einem zufälligen Trainingsvektor
39            Digit randomOne = digits.get(r.nextInt(digits.size()));
40
41            //Skalarprodukt mit der Näherung
42            double skalarprodukt = approx.transpose()
43                .times(randomOne.getMatrix()).get(0, 0);

```

```

43
44 //mit Lernfaktor auf die Approximation draufaddieren
45 // approx += (r-approx*skp)*skp*gamma
46 approx.plusEquals(randomOne.getMatrix().minus(
47     approx.times(skalarprodukt)
48     ).times(skalarprodukt*gamma)
49 );
50
51 //Lernfaktor verkleinern
52 gamma = gamma * beta / (beta+i);
53 i++;
54 }
55 gammaNull/=1.15;
56 } while (i!=15000); //wir sind nicht konvergiert, gamma war zu groß
57 return approx;
58 }

```

Damit wir den Algorithmus nochmal anwenden können, um die nächste Hauptkomponente zu finden, müssen wir die Daten in Richtung dieser Hauptkomponente zusammenquetschen. Das erledigt diese Funktion:

```

1 private static void squash(List<Digit> digits, Matrix pc) {
2     double skalar;
3     for(Digit d: digits) {
4         // d = d - <d,pc>*pc
5         skalar = d.getMatrix().transpose().times(pc).get(0,0);
6         d.setMatrix(d.getMatrix().minus(pc.times(skalar)));
7     }
8 }

```

Mit Hilfe des Skalarprodukts wird nur der zur Hauptkomponente orthogonale Anteil behalten. Mit diesen beiden Funktionen können wir uns eine Matrix aus den Hauptkomponenten bauen, die wir dann benutzen können, um die Dimensionalität der Daten zu reduzieren.

```

1 private static Matrix pCA(List<Digit> digits, int n, int dim) {
2     double[][] pcaMatrix = new double[n][dim];
3     for(int i=0;i<n;++i) {
4         Matrix pcI = getPC(digits,dim);
5         squash(digits,pcI);
6         pcaMatrix[i]=pcI.getRowPackedCopy();
7     }
8     return new Matrix(pcaMatrix);
9 }

```

Damit das ganze auch ordentlich funktioniert, müssen die Daten zentriert werden. Das wird vom Konstruktor des Oja-Objects erledigt.

Die Erkennungsrate haben wir wie gewohnt bestimmt

```

1 //k=#Hauptkomponenten
2 for (int k=start;k<=end;k+=step) {
3     int success = 0;
4
5     //i=Iterationen über die gemittelt wird
6     for (int i=0;i<iter;++i) {
7         digits = Helpers.getAllTrainingsDigitsFromDB();
8         testDigits = Helpers.getAllTrainingsDigitsFromDB();
9         Oja oja = new Oja(digits,192,k);
10        oja.reduce(digits); //dimensionen reduzieren
11        oja.reduce(testDigits);
12        for (Digit testDigit: testDigits) {
13            if (KNN.classifyDigit(testDigit,digits, 3, k) == testDigit.getLabel())
14                ++success;
15        }
16        System.out.print(".");
17    }
18    System.out.printf("\n%d_&_s\n", k, ((double)success/(testDigits.size()*iter)));
19 }

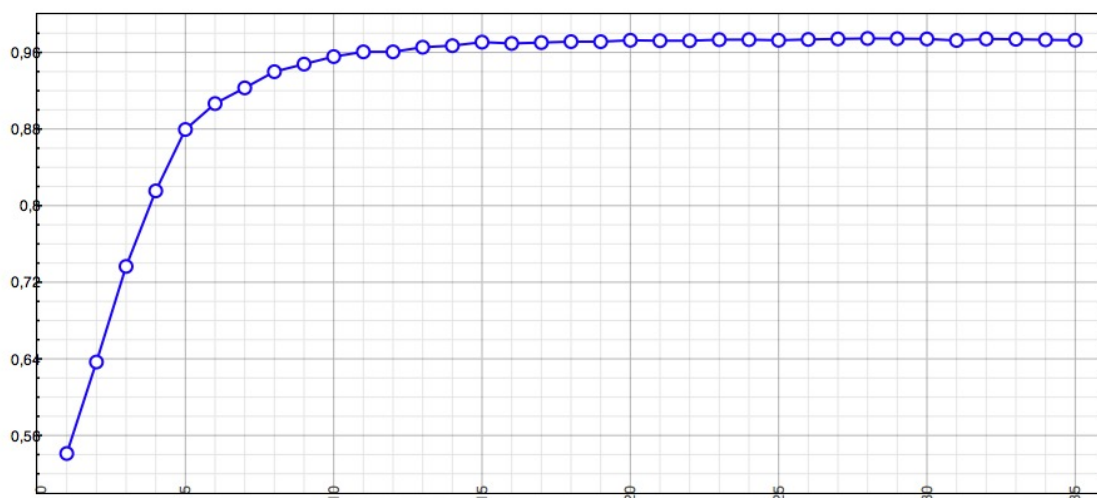
```

Auswertung

Während unserer Tests haben wir mit verschiedenen Werten für die Lernkonstanten gespielt. Es hat sich herausgestellt, dass γ am Anfang möglichst groß gewählt werden sollte, um gute Erkennungsraten zu erreichen. Ebenso hat ein sehr großes β die Erkennungsraten verbessert. Wir haben zu Testzwecken den Algorithmus auf einer sehr kleinen Menge 2D-Vektoren laufen lassen, die alle auf einer Linie lagen. Mit kleinen Werten für γ, β war die gefundene Hauptkomponente total falsch, vermutlich weil der Algorithmus zu schnell konvergiert ist.

Allerdings sorgen zu große Werte von γ dafür, dass der Algorithmus nicht mehr konvergiert (weil der Näherungsvektor nicht mehr die Länge 1 hat, sondern immer länger wird). Deswegen haben wir nach einer festen Zahl von Iterationen abgebrochen, wenn wir noch nicht konvergiert waren, und haben es mit einem kleineren γ nochmal probiert.

Mit den besten von uns gefundenen Werten ergab sich dieses Bild:



Hauptkomponenten	Erkennungsrate
1	0.5411211816045652
2	0.6366565961732125
3	0.7364887546156428
4	0.8153742866733803
5	0.8794897616649883
6	0.9065458207452165
7	0.9229271567640148
8	0.9398455857670359
9	0.9477677072843236
10	0.9556226921785834
11	0.9605908022826452
⋮	
32	0.9740852635112454
33	0.973749580396106
34	0.9731453507888553
35	0.9727425310506882

Gemittelt über 15 Durchläufe, kNN-k=3

Wie man sieht, ist der Algorithmus von Oja sehr gut darin, die relevanten Dimensionen zu liefern. Schon mit der ersten Hauptkomponente erreichen wir Erfolgsraten von über 50%, mit 10 Hauptkomponenten schon sagenhafte 95%. Mehr Dimensionen bringen nur noch marginale Gewinne, für sehr große Dimensionen nimmt die Erkennungsrate sogar wieder leicht ab (bei 192 Hauptkomponenten liegt sie nur noch bei etwa 93%).