

Mustererkennung: Übungsblatt 5

von

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

Aufgabe 1

In der Aufgabe soll das Verfahren der Fisher Diskriminante zur binären Klassifikation der Zifferndaten benutzt werden. Da die Zifferndatenbank sich nicht nur in zwei Klassen aufteilen lässt, benutzen wir einen Decision Directed Acyclic Graph, der jeweils nur zwei Klassen gegeneinander antreten lässt und durch die Auswertung der Fisher Diskriminanten eine Option ausschließen kann.

Wir müssen also zunächst mit Hilfe der Trainingsdaten diese Klassifikatoren trainieren. Dazu trennen wir zunächst unsere Trainingsdaten in die 10 Klassen auf und erzeugen dazu für alle zweielementigen Untermengen der Klassen die Klassifikatoren, in dem wir für unsere Klassifizierungsfunktionen $f(x) = w^T x + w_0$ jeweils w und w_0 bestimmen.

Dabei sagt uns die Fisher Diskriminante, wie w auszusehen hat, damit die beiden zu betrachtenden Klassen am besten zu trennen sind. Und zwar ergibt sich w als

$$w = S_w^{-1}(m_i + m_j)$$

wobei S_w die Addition der Scattermatrizen von Klasse i und j ist. Die Scattermatrix ist dabei definiert als

$$S_i = \sum_{x \in K_i} (x - m_i)(x - m_i)^T$$

```

1  public Matrix fisher(List<Digit> c1, List<Digit> c2) {
2      Matrix w = null;
3      int dim = c1.get(0).getFeatures().length;
4      Matrix sW = new Matrix(dim, dim);
5
6      // Scattermatrizen addieren zu S_w und Schwerpunkte berechnen
7      Digit balancePointC1 = calculateBalancePoint(c1);
8      Digit balancePointC2 = calculateBalancePoint(c2);
9      sW = calculateSW(c1, c2);
10
11     // Mittelpunkte aufsummieren
12     Matrix bpMatrix = new Matrix(dim, 1);
13     bpMatrix.plusEquals(balancePointC1.digitToMatrix());
14     bpMatrix.plusEquals(balancePointC2.digitToMatrix());
15
16     Matrix sWInvert;
17     try {
18         sWInvert = sW.inverse();
19     } catch (RuntimeException e) {
20         // kann nicht invertiert werden, daher approximieren wir.
21         Matrix identity = Matrix.identity(sW.getColumnDimension(), sW
22             .getRowDimension());
23         double lambda = Math.random() / 100;
24         sW.plusEquals(identity.times(lambda));
25         sWInvert = sW.inverse();
26     }
27     w = sWInvert.times(bpMatrix);
28     return w;
29 }
```

Genauso wird das auch in den Code umgesetzt. Zu beachten ist noch, dass es vorkommen kann, dass S_w^{-1} nicht invertierbar ist. Dann addieren wir zu S_w eine um einen ganz kleinen Faktor λ multiplizierte Einheitsmatrix und invertieren dann diese neue Matrix. Wir approximieren also die invertierte Matrix.

w_0 wird dann auch nach Schema f berechnet und gibt dabei sowas wie den Abstand der Mittelpunktprojektionen auf w an.

```

1  private Matrix calculateW0(List<Digit> c1, List<Digit> c2) {
2      Matrix w0;
3      Matrix sW = calculateSW(c1, c2);
4      Matrix sWInvert;
5      try {
6          sWInvert = sW.inverse();
7      } catch (RuntimeException e) {
8          // kann nicht invertiert werden, daher approximieren wir.
9          Matrix identity = Matrix.identity(sW.getColumnDimension(), sW
10             .getRowDimension());
11          double lambda = Math.random() / 100;
12          sW.plusEquals(identity.times(lambda));
13          sWInvert = sW.inverse();
14      }
15      Digit bp1 = calculateBalancePoint(c1);
16      Digit bp2 = calculateBalancePoint(c2);
17
18      Matrix sum = bp1.digitToMatrix().plus(bp2.digitToMatrix());
19      Matrix dif = bp1.digitToMatrix().minus(bp2.digitToMatrix());
20
21      w0 = (sum.transpose().times(sWInvert)).times(dif);
22      w0.timesEquals(-0.5);
23
24      return w0;
25  }

```

Wenn wir nun alle Klassifikatoren zusammen haben, können wir jedes Testdatum durch den DDAG jagen. Dabei lassen wir zunächst die kleinste und die größte Klasse (also vom Label her) gegeneinander antreten, d.h. wählen den zugehörigen Klassifikator aus, klassifizieren das Datum d anhand dessen. Wenn wir eine -1 zurückgeliefert bekommen, wissen wir, dass das Datum nicht zur ersten Klasse gehört, sondern eher zur zweiten. Daher schmeißen wir nun die erste Klasse aus dem Wettbewerb raus, in dem wir den Zähler für die kleinere Klasse hochsetzen. Dadurch wird in der verschachtelten Liste der Klassifikatoren diese Klasse nicht mehr ausgewählt und wir können die nächsten gegeneinander antreten lassen. Zum Schluss bleibt nur noch eine Klasse übrig, dieser wird dann das Datum zugeordnet.

```

1  public int oneVsOne(List<List<Classifier>> classifiers, Digit d) {
2      int i = 0;
3      int j = 9;
4
5      while (true) {
6          if (i == j) { // wir sind fertig, es ist in Klasse i
7              return i;
8          }
9
10         // Richtigen Klassifikator auswählen
11         Classifier c = classifiers.get(i).get(j - i - 1);
12         int classify = classify(c, d.digitToMatrix());
13         if (classify == -1) { // nicht Klasse 1, also Klasse 2 dabehalten
14             i++;
15         } else { // nicht Klasse 2, also Klasse 1 drinnenbehalten.
16             j--;
17         }
18     }
19 }

```

Um die Erkennungsrate zu bestimmen, iterieren wir dieses `oneVsOne`-Verfahren über alle Testdaten und zählen, wie oft wir richtig lagen.

```

1  // Testdaten klassifizieren
2  int right = 0;
3  for (Digit d : f.testDigits) {
4      int classify = f.oneVsOne(classifiers, d);
5      if (classify == d.getNumber()) {
6          right++;
7      }

```

```
8  
9     }  
10    System.out.println("Erkennungsrate:␣" + (double) right  
11                        / f.testDigits.size());
```

Als Erkennungsrate bekommt man dann etwas um die 22% heraus. Das ist natürlich relativ schlecht. Vermutlich ist der Fehler auf einen Berechnungsfehler bei den Klassifikatoren zurückzuführen. Es wurde z.B. bei der Berechnung von w_0 stark vereinfacht.