

Mustererkennung: Übungsblatt 6

von

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

Aufgabe 1

Die Aufgabe war es, je zwei Klassen unserer Trainingsmenge von Ziffern mit dem Perzeptron-Lernalgorithmus zu trennen, um dann mit einem DDAG und einem one-versus-one Klassifikator die Testdaten klassifizieren zu können.

Code

Wir benutzen ein Perzeptron-Objekt, in dessen Konstruktor wir die Vorverarbeitungsschritte durchführen. Zunächst werden die Test- und Trainingsdaten zentriert und auf Länge 1 normiert. Ursprünglich wurde hier auch noch die Dimension der Daten um 1 erhöht, um immer eine Trennung durch den Ursprung zu ermöglichen. Das hat allerdings eher einen negativen Einfluss auf die Erkennungsrate gehabt (durchschnittlich 2% weniger), sodass wir dies wieder rausgenommen haben. Wie sich herausstellt, findet man auch so immer eine optimale Trennung.

Anschließend wird die Trainingsmenge in die 10 Ziffernklassen aufgeteilt, sodass wir bei der Klassifizierung einer Testziffer je zwei Klassen gegeneinander antreten lassen und durch den Perzeptron-Klassifikator in jedem Schritt eine Klasse ausschließen können. Um später Rechenaufwand zu sparen, berechnen wir die Klassifikatoren für je zwei Klassen im Voraus und schreiben sie in eine Matrix. In dieser Matrix steht an Position (i, j) das **Classifier**-Objekt für Klasse i und Klasse j .

```

1  /**
2   * Erstellt eine n x n Matrix von Klassifikatoren. Man braucht nur die
3   * Hälfte der Matrix (ohne Diagonale) zu füllen, also nur n*(n-1)/2
4   * Einträge (mit n = #Klassen). Denn für einen Klassifikator f gilt:
5   * f_ij(x) = -f_ji(x) für alle 0 <= i < n und 0 <= j < n
6   * @return Klassifikator-Matrix
7   */
8  private Classifier[][] computeClassifiers() {
9      Classifier[][] classifiers = new Classifier[10][10];
10     for (int i = 0; i < classifiers.length; i++)
11         for (int j = i+1; j < classifiers[0].length; j++) {
12             classifiers[i][j] = perceptronLA(i, j);
13         }
14
15     return classifiers;
16 }

```

Ein **Classifier** kapselt den Trennungsvektor zweier Klassen N und P , den wir durch den Perzeptron-Lernalgorithmus finden. Dazu wählen wir zunächst zufällig einen Vektor aus $P \cup \bar{N}$ als ersten Trennungsvektor w . Dann nehmen wir wiederholt einen zufälligen Vektor x aus $P \cup N$ und überprüfen, ob unser bisheriges w diesen der richtigen Klasse zuordnen würde, d.h. das Skalarprodukt $w^T x$ muss größer 0 sein, wenn $x \in P$ und kleiner 0, wenn $x \in N$. Klassifizieren wir falsch, müssen wir unser w verändern und es entweder durch Addition von x , wenn $x \in P$ oder Subtraktion, wenn $x \in N$ in den positiven Halbraum verschieben.

```

1  /**
2   * Findet mit dem Perzeptron-Lernalgorithmus eine Trennung w zweier
3   * Klassen von Daten mit Index i und j.
4   * @param i Index der ersten Klasse
5   * @param j Index der zweiten Klasse

```

```

6  * @return Klassifikator, der w enthält
7  */
8  private Classifier perceptronLA(int i, int j) {
9      List<Digit> classP = classes.get(i);
10     List<Digit> classN = classes.get(j);
11     Matrix w = null;
12     Digit digit = null;
13
14     //zufälliges w aus negiertem N vereinigt P wählen
15     Random r = new Random();
16     int random = r.nextInt(classP.size()+classN.size());
17     if (random < classP.size()) w = classP.get(random).getAsMatrix();
18     else w = classN.get(random-classP.size()).getAsMatrix().uminus();
19
20     do {
21         for (int t = 0; t < 1000; t++) {
22             //wähle zufällig eine Zahl aus N vereinigt P
23             random = r.nextInt(classP.size()+classN.size());
24             if (random < classP.size()) digit = classP.get(random);
25             else digit = classN.get(random-classP.size());
26
27             //berechne Skalarprodukt
28             double scp = Helpers.scalarProduct(w, digit.getAsMatrix());
29             //überprüfen ob digit im richtigen Halbraum der durch w
30             //induzierten Teilung liegt
31             if (classP.contains(digit) && scp < 0)
32                 w.plusEquals(digit.getAsMatrix());
33             else
34                 if (classN.contains(digit) && scp > 0)
35                     w.minusEquals(digit.getAsMatrix());
36         }
37     } while(!isPartingGood(classP, classN, w)); //sind wir fertig?
38     return new Classifier(w);
39 }

```

Das Ganze machen wir nun so lange, bis wir die perfekte Trennung der Klassen finden, welches wir in der Methode `isPartingGood()` überprüfen. Die Trennung ist genau dann „gut“, wenn das Skalarprodukt von w und allen $x \in P \cup \bar{N}$ immer positiv ist. Sobald wir ein x finden, das falsch klassifiziert wird, gibt die Methode `false` zurück.

```

1  /**
2   * Überprüft, ob die momentane Trennung der Daten "gut" ist, d.h. alle
3   * Ziffern aus Klasse P müssen auf der einen Seite der durch w induzierten
4   * Trennungsebene liegen und alle Ziffern aus N auf der anderen.
5   * @param classP erste Ziffernklasse
6   * @param classN zweite Ziffernklasse
7   * @param w zu prüfender Trennungsvektor
8   * @return true, wenn die Trennung gut ist, sonst false
9   */
10 private boolean isPartingGood(List<Digit> classP, List<Digit> classN,
11     Matrix w) {
12     boolean isGood = false;
13     //bei Klasse P muss das Skalarprodukt mit w immer positiv sein (d.h.
14     //der Betrag des Winkels aller x aus P mit w ist <= 90), dann
15     //liegen alle x aus P auf der richtigen Seite der Trennungsebene.
16     for (Digit digit: classP) {
17         isGood = Helpers.scalarProduct(w, digit.getAsMatrix()) >= 0;
18         if (!isGood) return false;
19     }
20     //bei Klasse N muss das Skalarprodukt mit w immer negativ sein (d.h.
21     //der Betrag des Winkels aller x aus N mit w ist > 90), dann
22     //liegen alle x aus N auf der richtigen Seite der Trennungsebene.
23     for (Digit digit: classN) {
24         isGood = Helpers.scalarProduct(w, digit.getAsMatrix()) < 0;
25         if (!isGood) return false;
26     }
27     return true;
28 }

```

Haben wir auf diese Weise die Matrix von Klassifikatoren berechnet, geht die eigentliche Klassifizierung der Testdaten sehr schnell. Für alle Ziffern aus der Testmenge rufen wir die Methode `classifyOneVsOne()` auf.

```

1  /**
2   * Klassifiziert die übergebene Ziffer mithilfe eines DDAG und
3   * 1-versus-1 Klassifizierung.
4   * Es treten dabei immer zwei Klassen gegeneinander an. Der Perzeptron-
5   * Klassifikator bildet testDigit auf 1 oder -1 ab. 1 heißt, dass wir die
6   * zweite Klasse ausschließen können, -1, dass wir die erste Klasse
7   * ausschließen können. So laufen wir die Klassifizierungs-Matrix entlang
8   * und schließen in jedem Schritt eine Klasse aus, bis wir auf der
9   * Diagonalen landen. Das ist dann die zugehörige Klasse.
10  * @param testDigit Ziffer der Testmenge
11  * @return          Label der Ziffer
12  */
13  public int classifyOneVsOne(Digit testDigit) {
14      int i = 0, j = classes.size()-1;
15      while(i != j) {
16          double perzeptron = classifiers[i][j].classify(testDigit);
17          if (perzeptron == 1) j--;
18          else i++;
19      }
20      return i;
21  }

```

Testlauf

Wir führen die Klassifikation mehrmals durch und berechnen den Mittelwert der Erkennungsraten. Im Schnitt erhält man Erkennungsraten um die 90%.

Starte Iterationen ...

```

[i = 0] Erkennungsrate: 0.9045226130653267
[i = 1] Erkennungsrate: 0.9045226130653267
[i = 2] Erkennungsrate: 0.9045226130653267
[i = 3] Erkennungsrate: 0.8693467336683417
[i = 4] Erkennungsrate: 0.9045226130653267
[i = 5] Erkennungsrate: 0.9045226130653267
[i = 6] Erkennungsrate: 0.9095477386934674
[i = 7] Erkennungsrate: 0.8844221105527639
[i = 8] Erkennungsrate: 0.8793969849246231
[i = 9] Erkennungsrate: 0.9195979899497487
*****
ER ueber 10 Durchlaeufe: 0.8984924623115578

```