

Mustererkennung: Übungsblatt 1

von

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

Aufgabe 1

Zunächst ging es darum die Varianzen der $j = 192$ Merkmale der Ziffern aus der Datenbank zu bestimmen. Dazu haben wir die Formel wie folgt umgesetzt, wobei ein absteigend nach Varianz sortiertes Array von Tupeln (Merkmalsindex j , Varianz) zurückgegeben wird (Digit ist hierbei unsere eigene Klasse, die die eingelesenen Ziffern aus der Datenbank kapselt):

```

1  /**
2   * Berechnet die Varianz jedes Merkmals der mit n ausgewählten Ziffer aus
3   * der Trainingsdatenbank. Gibt ein nach Varianz aufsteigend sortiertes
4   * Array zurück.
5   * @param digits alle Trainingsziffern aus der Datenbank
6   * @return Array von Tupeln (Merkmalsindex, Varianz)
7   */
8  public static Tuple[] computeVariances(List<Digit> digits) {
9      double frac = 1.0/digits.size(); // 1/N
10     Tuple[] variances = new Tuple[digits.get(0).length()];
11     //für jedes Merkmal
12     for(int j = 0; j < digits.get(0).length(); j++) {
13         double sum = 0.0;
14         double mu = frac * sum(j, digits);
15         //für jede Ziffer aus der Trainingsmenge
16         for (int i = 0; i < digits.size(); i++) {
17             double temp = digits.get(i).getElement(j) - mu;
18             sum += temp * temp;
19         }
20         variances[j] = new Tuple(j, frac * sum);
21     }
22     Arrays.sort(variances);
23     return variances;
24 }
25
26 /**
27 * Berechnet die Summe alle Merkmale mit Index j
28 * @param j Merkmalsindex
29 * @param digits alle Trainingsziffern aus der Datenbank
30 * @return
31 */
32 private static double sum (int j, List<Digit> digits) {
33     double sum = 0.0;
34     for (int i = 0; i < digits.size(); i++)
35         sum += digits.get(i).getElement(j);
36     return sum;
37 }

```

a) Die Anordnung (2) sortiert die Features nach ihrer Varianz. Da wir Pixelwerte als unsere Features wählen, heißt das, dass die Pixel, die sich nie (bzw. selten) ändern (zum Beispiel die Ecken, die vermutlich immer weiß sind) weit hinten anordnen.

Mit Hilfe der Formel (3) werden diejenigen Features weggeworfen, die nur eine geringe Varianz haben (sie stehen hinten). Features, die sich in der gesamten Trainingsmenge nicht ändern, liefern nur wenig Entscheidungsinformationen; dass die Ecke links weiß ist, macht keine Aussage über die dargestellte

Zahl. Mit dieser Formel können also tendenziell unwichtige Dimensionen entfernt und die Berechnung der Nachbarn beschleunigt werden.

b) Zunächst haben wir die maximalen Dimensionen für alle gegebenen α -Werte nach Formel (3) berechnet.

```

1  /**
2   * Berechnet die maximale Dimension  $M(\alpha)$ , also die Anzahl der zu
3   * verwendenden Merkmale bei kNN.
4   * @param alpha
5   * @param variances Merkmalsvarianzen
6   * @return maximale Dimension im Bereich [0 .. Anzahl Merkmale]
7   */
8  public static int findMaxDimension(double alpha, Tupel[] variances) {
9      double sum = 0.0;
10     //die Summe aller Merkmalsvarianzen bilden
11     for(Tupel t : variances) sum += t.value;
12     double temp = 0.0;
13     for(int i = 0; i < variances.length; i++) {
14         temp += variances[i].value;
15         //wir sind über dem Schwellenwert gelandet
16         if (temp > alpha * sum) return i;
17     }
18     //for wurde nicht abgebrochen, wir wollen also alle features
19     return variances.length;
20 }

```

Nun können wir den k-NN Algorithmus mit den gerade bestimmten Anzahlen an Features ausführen. Dazu haben wir die Trainings- und Testdaten jeweils zeilenweise in ein Array eingelesen und in einem Digit-Objekt gekapselt, das sich auch das Label merken kann. Die Arrays aller Digit-Objekte sortieren wir nun genauso, wie die Varianzen es vorgeben (die Merkmale mit hoher Varianz stehen ganz vorne), damit man bei k-NN nur noch die Merkmale $0 - M(\alpha)$ betrachten muss. Nun können wir die k nächsten Nachbarn finden:

```

1  /**
2   * Findet die k nächsten Nachbarn zu testDigit, indem der Abstand von
3   * testDigit zu jeder Ziffer aus der Trainingsdatenbank berechnet wird
4   * und dann die k kleinsten ausgewählt werden. Zurückgegeben werden die
5   * Labels der k nächsten Nachbarn.
6   * @param k Anzahl der zu findenden nächsten Nachbarn
7   * @param features Anzahl der zu betrachtenden Features
8   * @param testDigit die zu klassifizierende Ziffer
9   * @param digits die Trainingsmenge von Ziffern
10  * @return Stimmen der k nächsten Nachbarn
11  */
12  public static int[] findKNearestNeighbors(
13      int k, int features, Digit testDigit, List<Digit> digits) {
14      Tupel[] distances = new Tupel[digits.size()];
15      int[] neighbors = new int[k];
16      for (int i = 0; i < digits.size(); i++)
17          distances[i] = new Tupel(i, squareDistance(testDigit, digits.get(i),
18              features));
19      Arrays.sort(distances); //Abstände (Tupel) werden absteigend sortiert
20      int j = 0;
21      //uns interessieren die letzten k Abstände, also die k kleinsten
22      for (int i = distances.length-1; i >= distances.length-k; i--)
23          neighbors[j++] = digits.get(distances[i].pos).getLabel();
24      return neighbors;
25 }

```

Für die Abstände zweier Ziffer, berechnen wir das Quadrat der Differenzen $d = (\vec{a} - \vec{b})^2$, wobei wir uns aus Effizienzgründen das Wurzelziehen sparen, denn wenn $d_1 > d_2 > 0$ dann gilt ja auch $\sqrt{d_1} > \sqrt{d_2}$.

Die Methode `findKNearestNeighbors()` gibt die Abstimmungsergebnisse (Labels) der k Nachbarn zurück, diese müssen nun nur noch ausgezählt werden.

```

1  /**
2  * Die Stimmen der Nachbarn werden ausgezählt und es wird die Zahl zurück
3  * gegeben, für die am häufigsten gestimmt wurde. Bei Gleichstand gewinnt
4  * die kleinste Zahl.
5  * @param neighbors Stimmen der Nachbarn
6  * @return Klassifizierungsergebnis
7  */
8  public static int vote(int[] neighbors) {
9      Integer[] histogram = new Integer[10];
10     //mit Nullen initialisieren
11     for (int i = 0; i < histogram.length; i++) histogram[i] = new Integer(0)
12     ;
13     //Vorkommen zählen
14     for (int n : neighbors)
15         ++histogram[n];
16     //den Index finden, an dem die größte Zahl steht, bei Gleichstand
17     //gewinnt die niedrigste Ziffer
18     return Helpers.maxIndex(histogram);
19 }

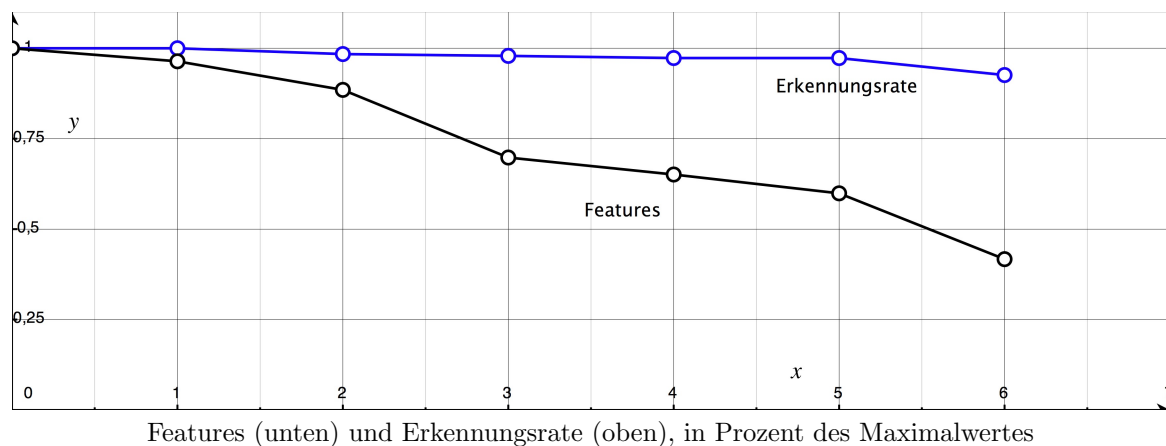
```

Mit diesem Verfahren klassifizieren wir nun alle Testdaten.

Auswertung In unserer experimentellen Durchführung hat sich die Vermutung aus a) bestätigt. Wie besonders gut an der Graphik zu erkennen ist, haben selbst starke Einschränkungen der Dimensionalität kaum einen Einfluss auf die Erkennungsrate. Diese folgenden Daten entstammen einem Durchlauf mit $k = 3$, das offenbar die besten Ergebnisse liefert.

α	$M(\alpha)$	Erfolge	Erkennungsrate
1.0	192	188	0.9447236
0.99	185	188	0.9447236
0.95	170	186	0.9296482
0.8	134	186	0.92462313
0.75	125	184	0.919598
0.7	115	182	0.919598
0.5	80	172	0.8743719

Die „krummen“ Erkennungsraten treten auf, da wir 199 gültige Testdaten eingelesen haben; ein Label war ungültig.



In einem nicht dokumentierten Durchlauf haben wir die Sortierung der Features umgekehrt, so dass Features mit besonders hoher Varianz weggelassen wurden. Für $\alpha = 0.5$ hat das sogar eine leichte Verbesserung der Erkennungsrate gebracht. Das ist erwartet, wenn die Daten nicht ganz rauschfrei sind. Bereiche mit hoher Varianz sind dann vermutlich Rauschen und somit wenig aussagekräftig für den Entscheidungsprozess.