Computerpraktikum Maschinelles Lernen

Thema 4 - Klassifikationsverfahren

Pascal Bauer, Raphael Millon, Florian Haas

Sommersemester 2020

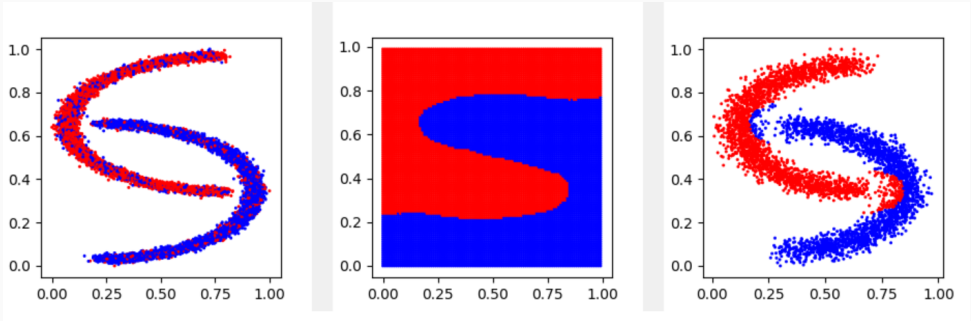
Table of contents

Die Klassifikationsergebnisse für *brute_sort* mit $k_{\text{max}} = 200, l = 5$:

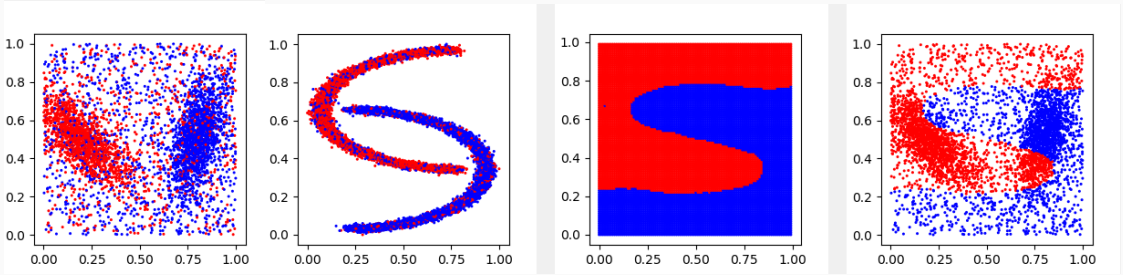
Datensatz:	Laufzeit (in Sekunden):	k*:	Fehlerrate:
australian	0.20	126	0.1346
bananas-1-2d	11.4	36	0.2083
bananas-1-4d	21.92	48	0.2088
bananas-2-2d	11.08	75	0.2122
bananas-2-4d	21.30	32	0.2213
bananas-5-2d	10.96	89	0.2555
bananas-5-4d	22.07	175	0.2542
cod-rna.5000	18.61	8	0.0693
ijcnn1	1150.36	1	0.0299
ijcnn1.10000	41.12	2	0.0247
ijcnn1.5000	10.01	2	0.0173
svmguide1	5.10	20	0.0343
toy-2d	11.35	100	0.2153
toy-3d	18.95	62	0.2288
toy-4d	21.89	39	0.2240
toy-10d	45.28	112	0.2140

Showcase Testen mit anderen Daten

Frage: Was passiert, wenn wir als Testdaten andere Datensätze verwenden?



(Von links nach rechts: Trainingsdaten (bananas-1-2d), Gitter, Ergebnis (mit Testdaten bananas-2-2d))



(Von links nach rechts: Testdaten (toy-2d), Trainingsdaten (bananas-1-2d), Gitter, Ergebnis)

Codebeispiele Struktur und Module

Code ist Open-Source auf Github:
`https://github.com/raphaelMi/computerpraktikum-maschinelles-lernen`

Unser Programm ist in folgende Module aufgeteilt:

- **main.py**: Hauptmodul mit wesentlichen Algorithmen
- **dataset.py**: Datensatz-Import/-Export
- **gui.py**: Grafische Oberfläche
- **kd_tree.py**: Hilfsmodul für k-d-Search
- **visual.py**: Plotting der Datensätze

Verwendete Bibliotheken:

- **numpy**: Effizientes (vektorisiertes) Rechnen
- **matplotlib**: Generieren der Plots
- **tkinter**: Grafische Benutzeroberflächen
- **scikit-learn**: Ein dritter Algorithmus zum Vergleich

Codebeispiele Klassifikation

Die **classify**-Funktion ist das "Herz" unseres Programmes:

```
def classify_gui(train_data, test_data, output_path, kset=K, l=5, algorithm='brute_sort'):
    if algorithm == 'brute_sort':
        dd, k_best = train_brute_sort(train_data, kset, l)
        print('k* =', k_best)
        f_rate, result_data = test(dd, test_data, k_best, output_path)
        return k_best, f_rate, result_data, dd
```

Parameter:

- **train_data**: Trainingsdaten
- **train_data**: Testdaten
- **output_path**: Ausgabedatei der Ergebnisdaten
- **kset**: Menge der k
- **l**: Partitionsanzahl
- **algorithm**: Suchalgorithmus für Nachbarn

Ablauf:

1. Training mit gegebenen Trainingsdaten und Sortieralgorithmus
2. Klassifikation und der Testdaten und Darstellung der Resultate

Codebeispiele Training

```python
def train_brute_sort(train_data, kset, l):
    # instead of making a random partition we use parts of a shuffled array
    # this results in disjoint sets d_i
    np.random.shuffle(train_data)
    # this way we have d_i = dd[i]
    dd = np.array_split(train_data, l)

    k_best_r = np.empty((l, len(kset)))
    for i, di in enumerate(dd):
        di_complement = np.concatenate(np.delete(dd, i, axis=0))# Complement of partition d_i

        for n, f in enumerate(f_train_brute_sort(di_complement, di[:, 1:], kset)):# Compute F_D_k function
            k_best_r[i][n] = R(di, stitch(f, di[:, 1:]))# Compute R_D_i
    k_best = kset[np.argmin(np.mean(k_best_r, axis=0))]# k*
    return dd, k_best
```

Ablauf:

1. Partitionierung des Datensatzes gemäß l
2. Klassifikation und der Testdaten und Darstellung der Resultate
3. Berechnung des $f_{D,k}(x)$ mittels brute_search
4. Berechnung der $\mathcal{R}_{D_i}$
5. Ermitteln des k^* über Minimierung des Mittelwertes