Computerpraktikum Maschinelles Lernen

Thema 4 - Binäre Klassifikation

Pascal Bauer, Raphael Million, Florian Haas

Sommersemester 2020

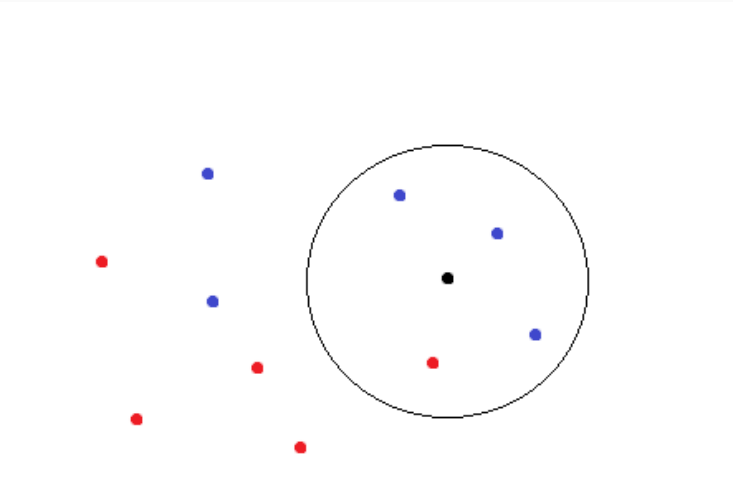
Table of contents

Ziel: Binäre Klassifikation eines zuvor unbekannten Datensatzes anhand Trainingsdaten.

Gegeben seien Trainingsdaten $(y_i, x_i)_{i=0}^{n}$ wobei $x_i \in [0,1]^d$ und $y_i \in \{-1, 1\}$. Wie ist es möglich für einen neuen Datensatz $(y_i', x_i')_{i=0}^{n'}$ die y_i' möglichst gut (mit minimaler Fehlerquote) zu schätzen?

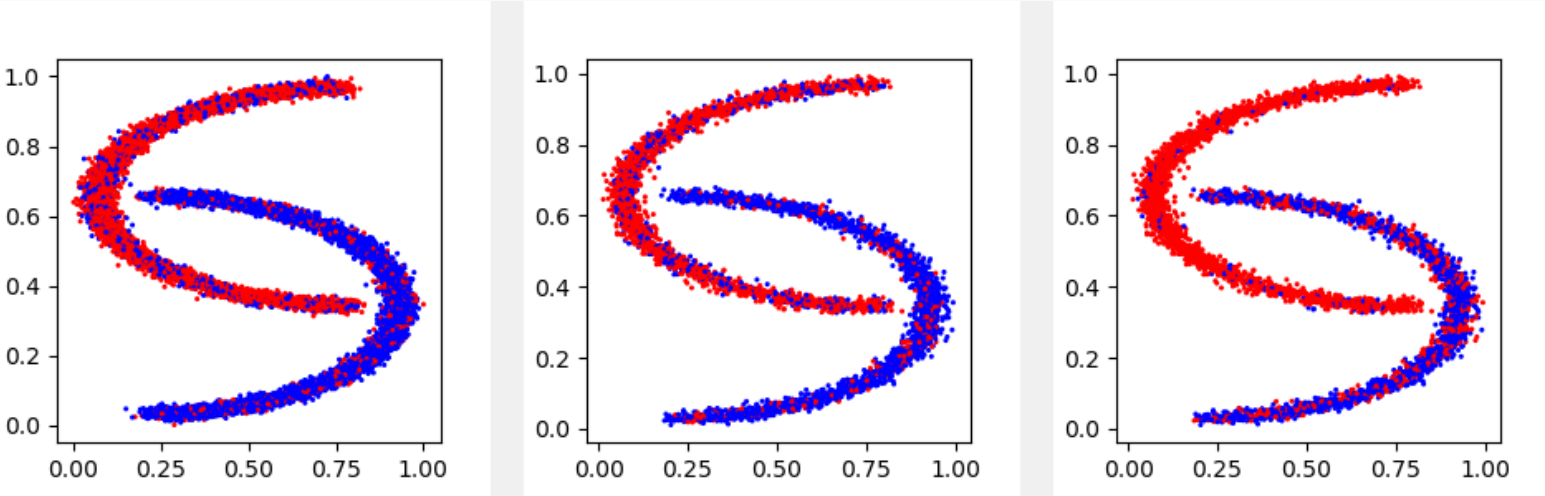
Gesucht ist also eine Funktion $f\colon [0,1]^d \to \{-1, 1\}$ die ein möglichst guter Prädiktor für neue Punkte sein soll.

$\Longrightarrow$ Es werden die k-nächsten Nachbarn des zu klassifizierenden Punktes x bestimmt und ein Majoriätsvotum durchgeführt.



Problem: Wie findet man man den optimalen Wert für k $(= k^*)$?

Keine gute Idee: Trainingsdaten gegen sich selbst testen und k^* als k mit kleinster Fehlerrate wählen



Mit diesem Ansatz würde immer $k^* = 1$ mit hoher Fehlerrate.

Benutzter Ansatz: Trainingsdatensatz D wird in l zufällige Teildatensätze D_i aufgeteilt. Teste D_i gegen Komplement $D \setminus D_i$ und wähle k^* als k mit durchschnittlich niedrigster Fehlerrate.

Theorie k-nächste Nachbarn

Die Bestimmung k-nächster Nachbarn ist Hauptkomplexität des Verfahrens. Es gibt zwei Möglichkeiten für die Bestimmung der nähesten Punkte:

1. **brute search** - Berechnen aller Abstände und Auswählen der k kleinsten. Dies hat asymtotische Zeitkomplexität $O(k \cdot n^2)$ und ist für dünne oder hoch dimensionsionale Datensätze effizient.
2. **tree search (etwa k-d oder ball tree)** - Ausnutzen der Baumeigenschaften um große Teile des Datensatzes zu eliminieren. Dies hat asymtotische Zeitkomplexität $O(k \cdot n \log n)$ und ist effizient falls $n \gg 2^d$ (worst case $O(k \cdot n^2)$).

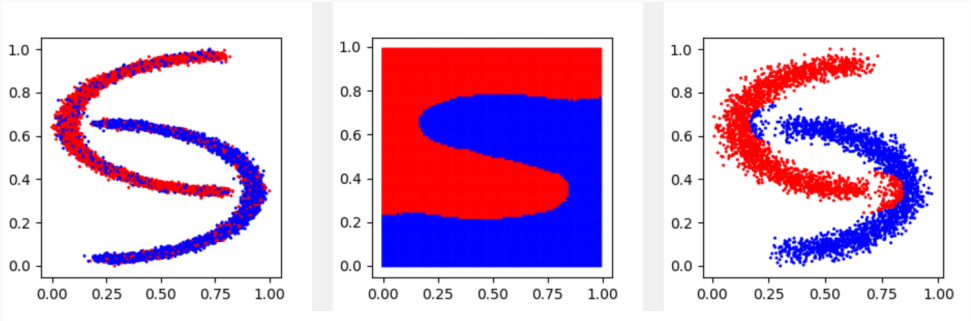
In unserem Programm sind beide Alogrithmen implementiert, der k-d tree Algorithmus ist aber nicht genügend optimiert.

Showcase

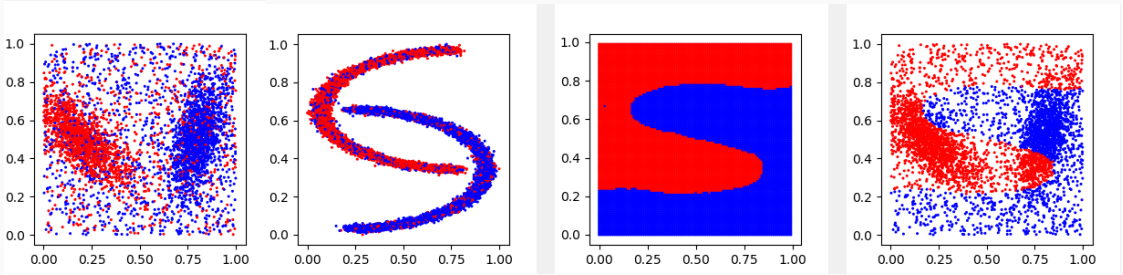
Die Klassifikationsergebnisse für *brute_sort* mit $k_{\text{max}} = 200, l = 5$:

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

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 der Testdaten und Darstellung der Resultate

Nun wird k^* ermittelt:

```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 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

Die Berechnung der $f_{D,k}(x)$ läuft wie folgt:

```python
# computes f_D,k for given x values for k in array shape
def f_train_brute_sort(data, x, kset):
    near = k_nearest_brute_sort(data, x, np.max(kset))  # using k_nearest to only compute it once
    y = data[:, :1]
    nearest_bin = np.take_along_axis(y, near, axis=0)  # assembles array of nearest ys
    results = []
    for k in kset:
        result = np.sign(np.sum(nearest_bin[:k], axis=0))
        result[result == 0] = 1  # sets sign(0) to 1
        results.append(result)
    return results
```

Ablauf:

1. Ermitteln der k-nächsten Nachbarn mittels brute_sort
2. Berechnung der $f_{D,k}(x)$ nach Vorschrift für alle k

Wir berechnen die k-nearest einmal für das größte k - **k_nearest_brute_sort** gibt die k-nearest nach Distanz sortiert zurück, sodass die übrigen k-nearest daraus abgeleitet werden können.

Algorithmus zur Ermittlung der k-nächsten Nachbarn.

```python
# return list of indices of nearest points
# x is in array-shape
def k_nearest_brute_sort(data, x, k_max):
    x_split = np.array_split(x, len(x) * len(data) // ARRAY_LIMIT + 1)
    indx = []
    for x in x_split:
        # dist = sp.distance.cdist(data[:, 1:], x, 'sqeuclidean')  #  very fast but scipy library not allowed
        cen = np.repeat(data[:, 1:][:, np.newaxis, :], len(x), axis=1) - x  # (~0.6 sec)
        sq = np.square(cen)  # (~0.2 sec)
        dist = np.sum(sq, axis=2)
        part_indx = np.argpartition(dist, k_max, axis=0)[:k_max]
        sort_indx = np.argsort(np.take_along_axis(dist, part_indx, axis=0), axis=0)
        indx.append(np.take_along_axis(part_indx, sort_indx, axis=0))
    return np.concatenate(indx, axis=1)
```

Ablauf:

1. Berechnung des euklidischen Abstandes aller Punkte zueinander
2. Sortierung nach Abstand pro Punkt
3. Herauspicken der k_{max} ersten (und damit nächsten) Punkte

Nun werden die Ergebnisse aus dem Training auf die Testdaten angewendet:

```
# compares prediction based on k* with test data and saves result_data
def test(dd, test_data, k_best,  output_path):
    compare = f_final(dd, test_data[:, 1:], k_best)
    result_data = stitch(compare, test_data[:, 1:])
    f_rate = R(test_data, result_data)
    print('Failure rate (compared to test data):', f_rate)
    dataset.save_to_file(output_path, result_data)

    return f_rate, result_data
```

Ablauf:

1. Assemblierung der finalen Funktion f_D gemäß Aufgabenstellung
2. Anwenden von f_D auf die Testdaten
3. Berechnen der Fehlerrate
4. Speichern des resultierenden Datensatzes

Codebeispiele Gitter

Anstatt f_D auf die Testdaten anzuwenden, wird f_D in Gitterpunkten ausgewertet:

```
# Only works for 2D plots
def grid(dd, k_best, grid_size):
    grid = [[n / grid_size, m / grid_size] for n in range(grid_size) for m in range(grid_size)]
    return stitch(f_final(dd, grid, k_best), grid)
```

Hier simulieren wir ein Gitter mit n gleichverteilten Punkten in $[0, 1] \times [0, 1]$.

Dies erlaubt uns, mehr Einsicht in f_D zu erhalten, als über die Plots der Ergebnisse möglich wäre.

Raum für Fragen und Diskussion...