ex2

May 3, 2019

0.1 Blatt 2

Übungen für Grundlagen Maschinelles Lernen

```
In [1]: from __future__ import print_function
        from IPython.display import display
        import numpy as np
        from numpy.linalg import inv
        import matplotlib
        import matplotlib.pyplot as plt
        import pandas as pd
        from collections import namedtuple
        from IPython.core.display import HTML
        plt.style.use('seaborn')
        %matplotlib inline
```

1 Task 2.1 Von Likelihood zum quadratischen Fehler:

Zeige durch explizite Rechnung, dass die Minimierung des $-LL(\omega)$ (negative log-Likelihood) auf den quadratischen Fehler führt (nehme an, der Logarithmus ist zur natürlichen Basis spezifiziert).

$$L = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y(\vec{x_n},\omega)-t_n)^2 \frac{1}{2\sigma^2}} \tag{1}$$

$$= \prod_{n=1}^{N} \sqrt{\frac{\beta}{2\pi}} e^{-(y(\vec{x_n},\omega)-t_n)^2 \frac{\beta}{2}} \tag{2}$$

$$= \left(\frac{\beta}{2\pi}\right)^{\frac{N}{2}} \prod_{n=1}^{N} e^{-(y(\vec{x_n},\omega)-t_n)^2 \frac{\beta}{2}} \tag{3}$$

$$\leftrightarrow LL = \frac{N}{2}(\ln\beta - \ln 2\pi) + \sum_{n=1}^{N} -(y(\vec{x_n},\omega)-t_n)^2 \frac{\beta}{2} \tag{4}$$

$$= \frac{N}{2}(\ln\beta - \ln 2\pi) + \frac{\beta}{2}\sum_{n=1}^{N} -(y(\vec{x_n},\omega)-t_n)^2 \tag{5}$$

$$\leftrightarrow -LL = \frac{\beta}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega)-t_n)^2 + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi \tag{6}$$

$$= \frac{\beta}{2}E(\omega) + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi \tag{7}$$

$$\tag{8}$$

Da argmin invariant gegenüber Skalierung und Verschiebung ist:

$$\omega_{ML} = \underset{\omega}{\operatorname{argmin}}\, E(\omega)$$

Zeige weiter, dass für die optimalen ω_{ML} dann für die Minimierung bzgl. β gilt:

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2$$

$$\beta_{ML} = \underset{\beta}{\operatorname{argmin}} - LL \tag{9}$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{\beta}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 + \frac{N}{2}\ln\beta - \frac{N}{2}\ln 2\pi \tag{10}$$

$$= \underset{\beta}{\operatorname{argmin}} \frac{\beta}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 + \frac{N}{2}\ln\beta \tag{11}$$

$$\tag{12}$$

$$\rightarrow \nabla_{\beta_{ML}} \frac{\beta_{ML}}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 + \nabla_{\beta_{ML}} \frac{N}{2}\ln\beta_{ML} = 0 \tag{13}$$

$$\leftrightarrow \frac{1}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 + \frac{N}{2}\frac{1}{\beta}_{ML} = 0 \tag{14}$$

$$\leftrightarrow \frac{N}{2}\frac{1}{\beta}_{ML} = \frac{1}{2}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 \tag{15}$$

$$\leftrightarrow \frac{1}{\beta}_{ML} = \frac{1}{N}\sum_{n=1}^{N}(y(\vec{x_n},\omega) - t_n)^2 \tag{16}$$

$$\tag{17}$$

2 Task 2.2 Funktionsapproximation:

Hilfsfunktionen:

```
In [2]: def design_mat_polynomial(x, m):
            phi_x_T = np.array([
                np.power(x.ravel(), order) for order in range(m+1)])
            return phi_x_T.T

        def get_y(w, x):
            w = np.array(w)
            m = w.shape[0] - 1
            phi_x = design_mat_polynomial(x, m)
            y = phi_x @ w
            return y

        def plot_weights(weights, label, x_min, x_max, ax):
```

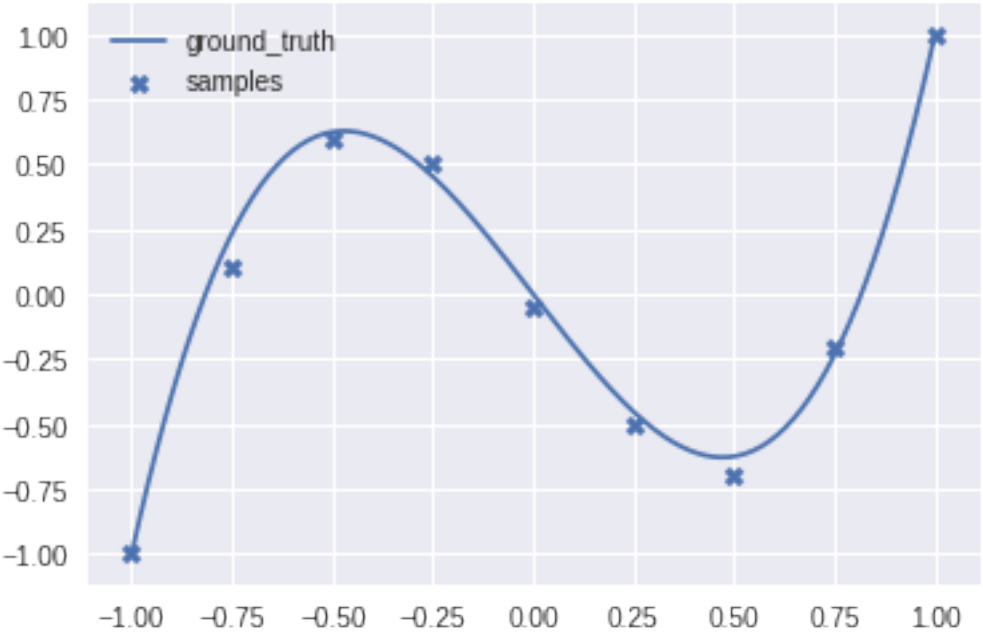
```
        n_samples = 200
        xs = np.linspace(x_min, x_max, n_samples)
        ys = get_y(weights, xs)
        ax.plot(xs, ys, label=label)

    def plot(df, named_weights):
        ax = plt.gca()
        cols = df.columns
        assert 'x' in cols, 'df must contain column \'x\''
        if 'y' in cols:
            ax.scatter(df.x, df.y, label='samples', marker='X')
        if weights:
            for label, w in named_weights.items():
                plot_weights(w, label, min(df.x), max(df.x), ax)
        ax.legend()
```

Gegeben sie folgender Datensatz `df`, der durch Addition von Rauschen zur Funktion $y = 3x^3 - 2x$ generiert wurde:

```
In [3]: # visualize dataset
        weights = {}
        weights['ground_truth'] = [0, -2, 0, 3]
        data = np.array([
            [-1, -3/4, -1/2, -1/4, 0, 1/4, 1/2, 3/4, 1],
            [-1,  0.1,  0.6,  0.5, -0.05 ,-0.5, -0.7, -0.2, 1]
        ])
        n = data.size
        df = pd.DataFrame(data.T, columns=['x', 'y'])
        print("Datensatz:")
        display(df.T) # transpose to save some space
        plot(df, weights)
```

```
Datensatz:
```

	0	1	2	3	4	5	6	7	8
x	-1.0	-0.75	-0.5	-0.25	0.00	0.25	0.5	0.75	1.0
y	-1.0	0.10	0.6	0.50	-0.05	-0.50	-0.7	-0.20	1.0



2.0.1 Polinomial Fit

Ich defiere $\mathbf{X}$ als die polynomielle Designmatrix m-ten Grades von $\vec{x} = [x_0, ..., x_n]^T$:

$$\mathbf{X} = \begin{pmatrix} x_1^0 & x_1^1 & \cdots & x_1^m \\ \vdots & \vdots & \ddots & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{pmatrix}$$

Dann gilt für ein perfektes $\vec{w}$:

$$\vec{y} = \mathbf{X}\vec{w} \tag{18}$$
$$\leftrightarrow \mathbf{X^{-1}}\vec{y} = \mathbf{X^{-1}X}\vec{w} \tag{19}$$
$$\leftrightarrow \mathbf{X^{-1}}\vec{y} = \vec{w} \tag{20}$$

Jedoch ist $\mathbf{X}$ allgemein nicht invertierbar (nichteinmal quadratisch).
Stattdessen nehme ich die Moore-Penrose-Pseudoinverse $\mathbf{X}^+$

$$\mathbf{X}^+ = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

Damit ergibt sich

$$\vec{w}^* = \mathbf{X}^+\vec{y}$$
$$\vec{y}^* = \mathbf{X}\vec{w}^*$$

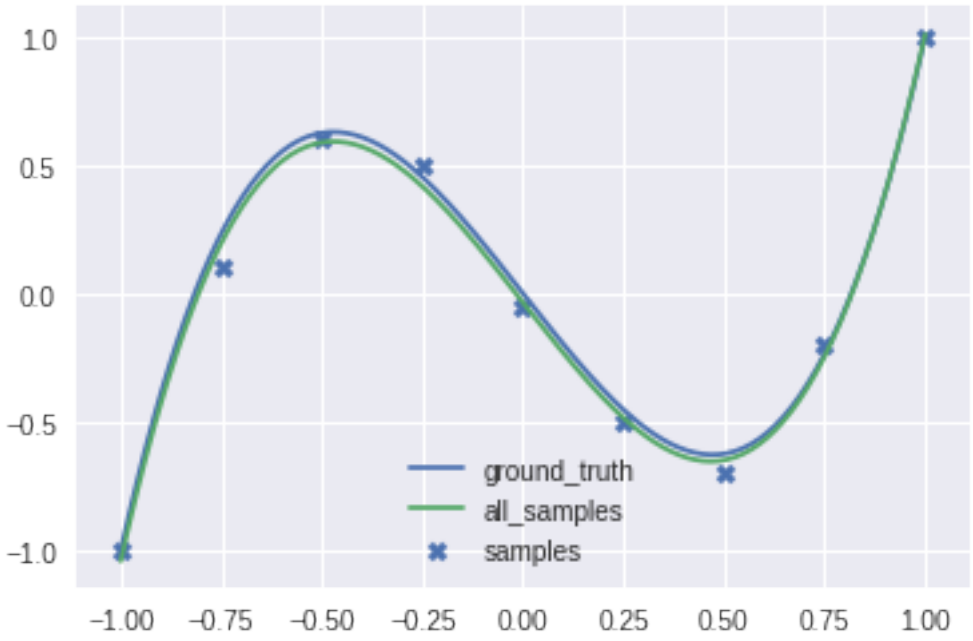
Wobei $\vec{y}^*$ die Projektion von $\vec{y}$ auf den Spaltenraum von $\mathbf{X}$ ist und somit den quadratischen Fehler $\|\vec{y} - \vec{y}^*\|^2$ minimiert.

```
In [4]: def pseudoinverse(A):
            return inv(A.T @ A) @ A.T

        def polyfit(x, y, m):
            x = np.array(x).reshape(-1, 1)
            y = np.array(y).reshape(-1, 1)
            phi_x = design_mat_polynomial(x, m)
            pinverse = pseudoinverse(phi_x)
            w = (pinverse @ y).reshape([-1, 1])
            return w
```

Verwende zur Approximation ein Polynom vom Grad $M = 3$. Was erwarten Sie für die Werte der Parameter w_0 bis w_3 ?

Gute Ergebnisse, da M dem tatsächlichen Grad des Datenmodells entspricht, und (für das geringe Rauschen) relativ viele Daten vorliegen

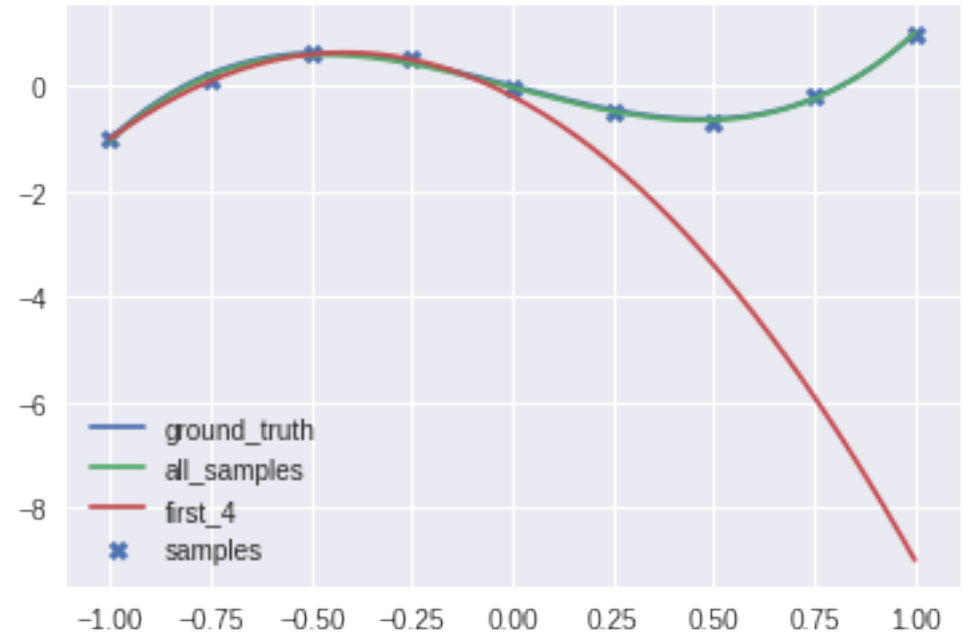
```
In [5]: weights['all_samples'] = polyfit(df.x, df.y, 3)
        plot(df, weights)
```



Approximieren Sie die Funktion indem Sie nur die Werte 1-4 der Tabelle verwenden. Was erwarten Sie jetzt für die Parameter?

Ich erwarte einen perfekten Fit für die ersten vier Punkte, da ein Polynom 3. Grades durch vier Datenpunkte beschreiben werden kann. Jedoch wird das Modell schlecht auf die anderen Daten generalisieren

```
In [6]: weights['first_4'] = polyfit(df[:4].x, df[:4].y, 3)
        plot(df, weights)
```

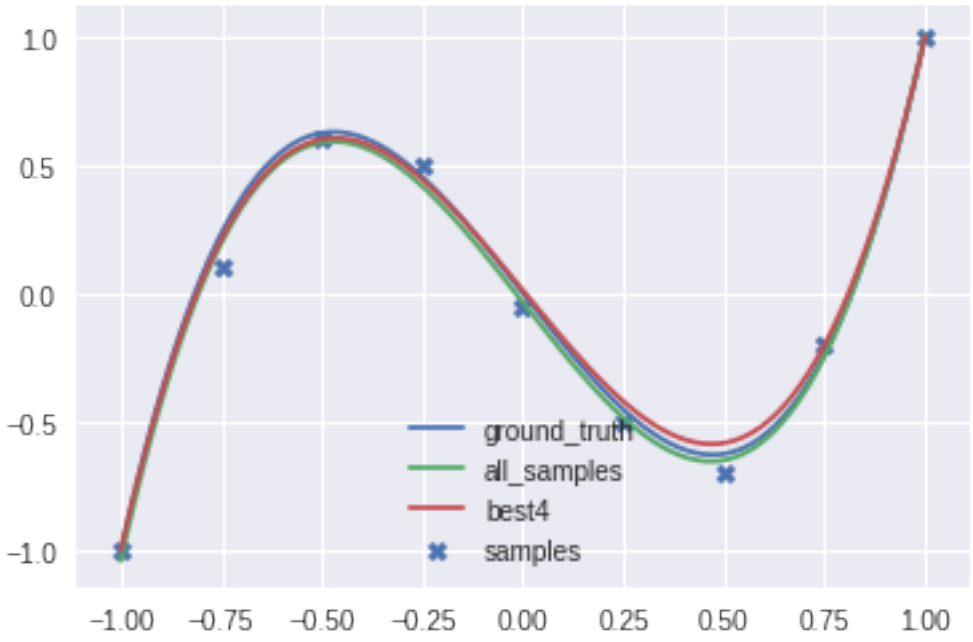


Berechnen Sie die Werte für w_i. Für welche Wahl von vier Punkten erwarten Sie die beste Approximation ?

```
In [7]: # get the best 4 points (least squared distance to y_true)
        df['y_gt'] = get_y(weights['ground_truth'], df.x)
        df['abs_err'] = np.abs(df.y - df.y_gt)
        df_sorted = df.sort_values(by='abs_err')
        df_best4  = df_sorted.head(4)
        display(df_best4)
```

	x	y	y_gt	abs_err
0	-1.00	-1.0	-1.000000	0.000000
8	1.00	1.0	1.000000	0.000000
2	-0.50	0.6	0.625000	0.025000
7	0.75	-0.2	-0.234375	0.034375

```
In [8]: # check the resulting plot
        weights.pop('first_4', None)
        weights['best4'] = polyfit(df_best4.x, df_best4.y, 3)
        df['y_best4'] = get_y(weights['best4'], df.x)
        plot(df, weights);
```



Berechnen Sie die Ausgaben des Datenmodells für die anderen 5 Werte und den entsprechenden Generalisierungsfehler.

```
In [9]: df_test = df.loc[~df.x.isin(df_best4.x)]
        df_test = df_test[['y', 'y_best4']]
        display(df_test)
        generalization_err = (df_test.y - df_test.y_best4)**2
        mse = generalization_err.sum() / generalization_err.count()
        print('mean squared error on test data: {:.4f}'.format(mse))
```

```
      y   y_best4
1  0.10  0.210000
3  0.50  0.442857
4 -0.05  0.011429
5 -0.50 -0.421429
6 -0.70 -0.582857
```

```
mean squared error on test data: 0.0078
```

Schätzen Sie schließlich die Varianz des Rauschens, mit dem die Daten generiert wurden.
The sample variance is calculated by the following formular:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \overline{Y}\right)^2$$

Die geschätzte Varianz ist die Varianz zwischen den Datenpunkten $y_{\vec{data}}$ und der besten Schätzung der Datenpunkte $\vec{y^*}$

```
In [10]: y_predicted = get_y(weights['all_samples'], df.x).ravel()
         variance = ((df.y - y_predicted) ** 2).sum() / n
         print('variance: {:.4f}'.format(variance))
         print('sigma: {:.4f}'.format(np.sqrt(variance)))
```

```
variance: 0.0013
sigma: 0.0360
```

3 Task 2.3 Lösung für lineare Modelle

Ein lineares Datenmodell ist linear in den Parametern ω, kann aber nichtlinear in den Eingaben sein. Es hat die generelle Form

$$y(x_n, \omega) = \sum_{m=0}^{M} \omega_m \Phi_m(x_n) = \vec{\omega}^T \mathbf{\Phi}(x_n)$$

$$\mathbf{\Phi}(x_n) = (\Phi_0(x_n), \ldots, \Phi_M(x_n))^T$$

wobei Φ_i eine beliebige (nichtlineare) Funktion der Eingaben sein kann, **die nicht von den Parametern abhängt!** Berechnen Sie den Gradienten der Fehlerfunktion $E(\omega)$ für ein lineares Datenmodell, wobei

$$E(\vec{\omega}) = \sum_{n=1}^{N}(t_n - \vec{\omega}^T \mathbf{\Phi}(x_n))^2$$

Setzen Sie den Gradienten zu Null (Bedingung für das Minimum !) und lösen Sie die entstehende Gleichung.

Zunächst schreibe ich die Fehlerfunktion um:

$$E(\vec{\omega}) = \sum_{n=1}^{N}(t_n - \mathbf{\Phi}(x_n)^T \vec{\omega})^2$$

Sei $\mathbf{X}$ die Designmatrix:

$$\mathbf{X} = \begin{pmatrix} \mathbf{\Phi}(x_1)^T \\ \vdots \\ \mathbf{\Phi}(x_n)^T \end{pmatrix}$$

... und $\vec{e}$ der n-dimensionale Fehlervektor:

$$\vec{e} = \vec{t} - \mathbf{X}\vec{\omega}$$

Dann lässt sich die Fehlerfunktion schreiben als:

$$\begin{aligned} E(\vec{\omega}) &= (\vec{e})^2 && (21)\\ &= \vec{e}^T\vec{e} && (22)\\ &= \left(\vec{t} - \mathbf{X}\vec{\omega}\right)^T \left(\vec{t} - \mathbf{X}\vec{\omega}\right) && (23)\\ &= \vec{t}^T\vec{t} - \vec{t}^T\mathbf{X}\vec{\omega} - \left(\vec{t}^T\mathbf{X}\vec{\omega}\right)^T + \vec{\omega}^T\mathbf{X}^T\mathbf{X}\vec{\omega} && (24)\\ &= \vec{t}^T\vec{t} - 2\left(\vec{t}^T\mathbf{X}\vec{\omega}\right) + \vec{\omega}^T\mathbf{X}^T\mathbf{X}\vec{\omega} && (25)\\ & && (26) \end{aligned}$$

Regeln zur Matrixdifferentialrechnung:

$\nabla_{\vec{x}}\mathbf{A} = 0$

$\nabla_{\vec{x}}\mathbf{A}\vec{x} = \mathbf{A}$

$\nabla_{\vec{x}}\vec{x}\mathbf{A} = \mathbf{A}^T$

$\nabla_{\vec{x}}\vec{x}^T\mathbf{A}\vec{x} = 2\,\vec{x}^T\mathbf{A}$

Gradient gleich 0 setzen:

$$\begin{aligned} 0 = \nabla_{\vec{\omega}} &= 0 - 2\,\vec{t^T}\mathbf{X} + 2\,\vec{\omega^T}\mathbf{X}^T\mathbf{X} && (27)\\ \rightarrow 2\,\vec{t}^T\mathbf{X} &= 2\,\vec{\omega}^T\mathbf{X}^T\mathbf{X} && (28)\\ \leftrightarrow 2\,\vec{t}^T\mathbf{X} &= 2\,\vec{\omega}^T\mathbf{X}^T\mathbf{X} && (29)\\ \leftrightarrow \vec{t}^T\mathbf{X} &= \vec{\omega}^T\mathbf{X}^T\mathbf{X} && (30)\\ \leftrightarrow \vec{\omega}^T &= \vec{t}^T\mathbf{X}\left(\mathbf{X}^T\mathbf{X}\right)^{-1} && (31)\\ \leftrightarrow \vec{\omega} &= \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{t} && (32)\\ & && (33) \end{aligned}$$