Blatt 3 Beispiellösung

Jonas Sitzmann

May 8, 2019

Ansehen auf Github oder mit Google Colaboratory ausführen.
Clone repository if necessary (for example for execution in google colab)

```
In [1]: import os
        if not os.path.exists('grundlagen_ml'):
          print('cloning repository github.com/jonasitzmann/grundlagen_ml')
          os.system('git clone https://github.com/jonasitzmann/grundlagen_ml')
          os.chdir('grundlagen_ml')
```

1 Task 3.1 Gewichteter Fehler theoretisch:

Nehmen Sie an, ein Datensatz $\vec{x}_n$, $\vec{t}_n$ ist gegeben, wobei jeder Trainingsdatenpunkt n mit einem Faktor r_n gewichtet ist. Setze ein lineares Model $y_n = \omega^T \phi(\vec{x}_n)$ an (für irgendein $\phi()$ und $n = 1 \dots N$ Trainingsdaten) und betrachte die entsprechende Fehlerfunktion.

$$E_D(\vec{\omega}) = \sum_{n=1}^{N} r_n \left(t_n - \vec{\omega}^T \phi\left(x_n\right) \right)^2$$

Finde die Lösung für die optimalen Parameter ω

Idee: Ich führe den gewichteten quadratischen Fehler auf den ungewichteten quadratischen Fehler zurück. Dieser lässt sich, wie in Übung 2 behandelt durch die Moore-Penrose-Pseudoinverse minimieren.

$$\begin{aligned}
E_D(\vec{\omega}) &= \sum_{n=1}^{N} r_n \left(t_n - \vec{\omega}^T \phi\left(x_n\right) \right)^2 && (1)\\
&= \sum_{n=1}^{N} \left(\sqrt{r_n} \left(t_n - \vec{\omega}^T \phi\left(x_n\right) \right) \right)^2 && (2)\\
&= \sum_{n=1}^{N} \left(\sqrt{r_n} t_n - \sqrt{r_n} \vec{\omega}^T \phi\left(x_n\right) \right)^2 && (3)\\
&= \sum_{n=1}^{N} \left(\sqrt{r_n} t_n - \sqrt{r_n} \phi\left(x_n\right)^T \vec{\omega} \right)^2 && (4)\\
& && (5)
\end{aligned}$$

Ich definiere $\tilde{t}_n$ und $\tilde{\phi}(x_n)$ wie folgt:
$\tilde{t}_n := \sqrt{r_n} t_n$
$\tilde{\phi}(x_n) := \sqrt{r_n}\phi(x_n)$
und erhalte:

$$E_D(\vec{\omega}) = \sum_{n=1}^{N} \left(\tilde{t_n} - \tilde{\phi}(x_n)^T \vec{\omega} \right)^2$$

Es müssen also lediglich alle Einträge t_n und $\phi(x_n)$ mit $\sqrt{r_n}$ multipliziert werden, bevor der quadratische Fehler minimiert wird.
Wie in Übung 2 ist $\mathbf{X}$ die Designmatrix.
Zusätzlich führe ich die Matrix $\mathbf{R}$ ein, als Diagonalmatrix mit den Einträgen $\sqrt{r_n}$ ein:

$$\mathbf{R} = \begin{pmatrix} \sqrt{r}_1 & & 0 \\ & \ddots & \\ 0 & & \sqrt{r}_n \end{pmatrix}$$

Aus Übung 2 ist die Pseudoinverse bekannt, durch die der ungewichtete quadratische Fehler minimiert wird:

$$\vec{\omega}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\vec{t}$$

Um die Parameter $\vec{\omega}^*_{\vec{r}}$ zu finden, die dengewichteten quadratischen Fehler minimieren, muss ich lediglich alle Einträge t_n und $\phi(x_n)$ mit $\sqrt{r_n}$ multiplizieren:

$$\vec{\tilde{t}} := \mathbf{R}\vec{t}$$

(Vectorpfeil wird im Folgenden weggelasen)

$$\tilde{\mathbf{X}} := \mathbf{R}\mathbf{X}$$

Damit gilt:

$$\vec{\omega}^*_{\vec{r}} = (\tilde{\mathbf{X}}^T\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^T\tilde{t}$$

und interpretiere die Parameter r_n im Sinne von (1) datenabhängiger Unsicherheit

Wenn Unsicherheiten als Varianten σ_n^2 bzw. als Präzisionen $\beta_n = \frac{1}{\sigma_n^2}$ der Datenpunkte bekannt sind, kann β_n als Gewicht verwendet werden. Somit haben sichere Datenpunkte einen höheren Einfluss auf $\vec{\omega}^*$.

Duplikationen von Datenpunktion

Duplizierte Datenpunkte, entsprechend doppelt gewertet werden sollen, können durch einen Datenpunkt ersetzt werden, dessen Gewicht doppelt so hoch ist. Das hilft, den Datensatz zu komprimieren.

2 Task 3.2 Gewichteter Fehler praktisch:

Verwenden Sie nun den Datensatz der Regressionsaufgabe von Blatt zwei und gewichten Sie die Punkte bei input $-\frac{1}{2}$ und $\frac{1}{2}$ doppelt, 10-fach, 100-fach. Dazu können Sie das publizierte Matlab-Script leicht verändern.

```
In [2]: import pandas as pd
        import numpy as np
        from numpy import sqrt, diag
        from functools import partial
        from helpers import load_dataset, plot, plot_df, design_mat_polynomial, get_y, plot_weig
        from numpy.linalg import pinv as pseudoinverse # better precision than my implementation
        from scipy.stats import norm # pdf for normal distribution
```

```
In [3]: """ returns weight vector with all ones except the indices of x that are in selection. "
        def get_weights(x, selection, selection_weight):
            return selection_weight**x.isin(selection)
        df = load_dataset()
        for factor in [1, 2, 10, 100]:
            df['weight_x{}'.format(factor)] = get_weights(df.x, [-.5, .5], factor)
        display(df.T)
```

	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
weight_x1	1.0	1.00	1.0	1.00	1.00	1.00	1.0	1.00	1.0
weight_x2	1.0	1.00	2.0	1.00	1.00	1.00	2.0	1.00	1.0
weight_x10	1.0	1.00	10.0	1.00	1.00	1.00	10.0	1.00	1.0
weight_x100	1.0	1.00	100.0	1.00	1.00	1.00	100.0	1.00	1.0

Zusätzlich verschiebe ich den Datenpunkt bei $x = -\frac{1}{2}$ um 0.3, damit der Einfluss der Gewichte gut sichtbar ist.

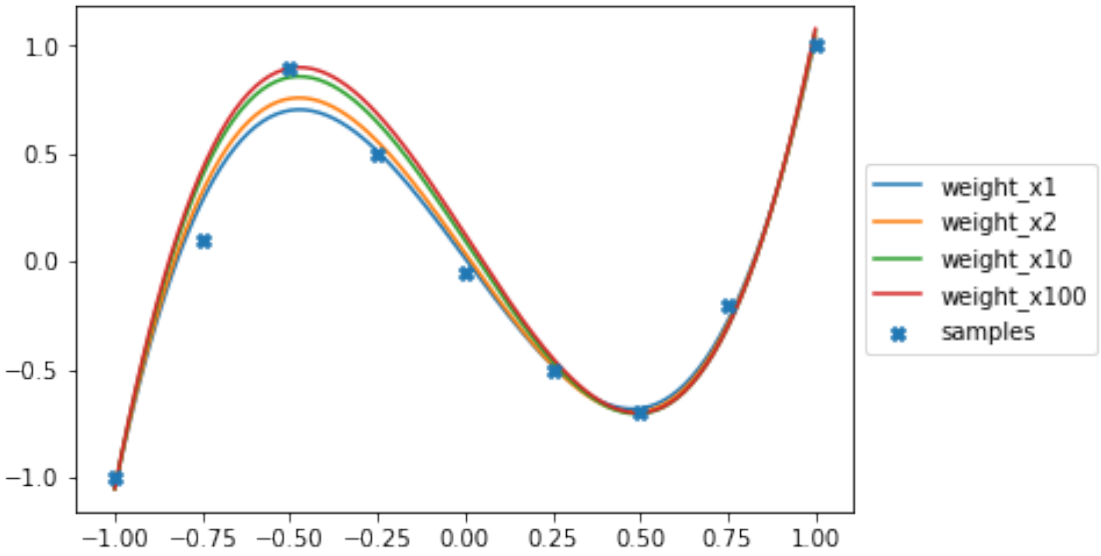
```
In [4]: df.loc[df.x==-1/2, 'y'] += 0.3
        display(df.T)
```

	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	0.9	0.50	-0.05	-0.50	-0.7	-0.20	1.0
weight_x1	1.0	1.00	1.0	1.00	1.00	1.00	1.0	1.00	1.0
weight_x2	1.0	1.00	2.0	1.00	1.00	1.00	2.0	1.00	1.0
weight_x10	1.0	1.00	10.0	1.00	1.00	1.00	10.0	1.00	1.0
weight_x100	1.0	1.00	100.0	1.00	1.00	1.00	100.0	1.00	1.0

```
In [5]: def polyfit(x, y, order=3, weights=None):
            x, y = [np.array(a).reshape(-1, 1) for a in [x, y]]
            weight_mat = np.eye(len(x)) if weights is None else diag(sqrt(np.array(weights)))
            design_mat = weight_mat @ design_mat_polynomial(x, order)
            y = weight_mat @ y
            params = pseudoinverse(design_mat) @ y
            return params.reshape(-1, 1)
```

```
In [6]: polyfit_weighted = partial(polyfit, df.x, df.y, order=3)
        w_cols = [col for col in df.columns if 'weight' in col]
```

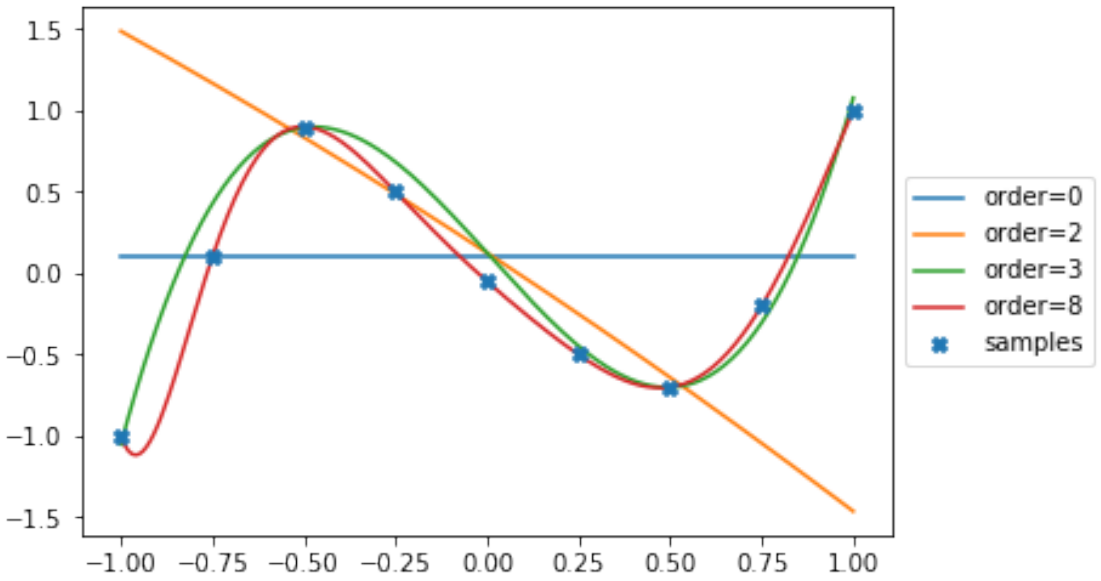
```
coeff_dict = {col : polyfit_weighted(weights=df[col]) for col in w_cols}
plot_df(df, coeff_dict)
```



Verwenden Sie wieder Polynome zur Approximation und variieren Sie den Grad. Was erwarten Sie, wie wird sich das optimale Modell mit der Wahl des Polynomgrades ändern wird?

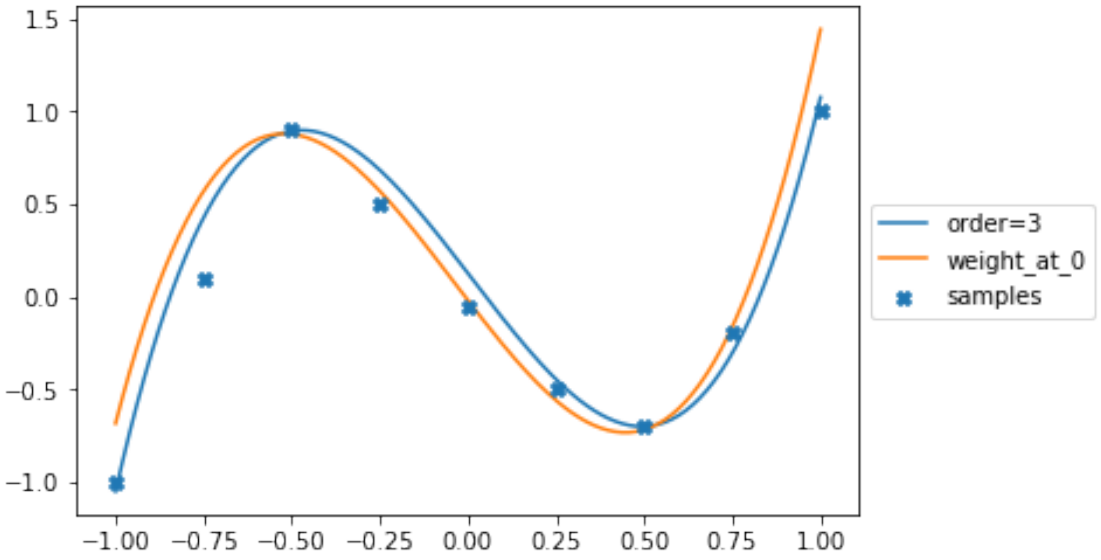
Je höher der Grad, desto kleiner die Abweichung zu den Datenpunkten. Allerdings führt ein hoher Grad zu kleinen "Schlenkern" und vielen Extrempunkten, generalisiert also schlecht.

```
In [7]: orders = [0, 2, 3, 8]
        order_coeff_dict = {"order={}".format(o) : polyfit(df.x, df.y, order=o, weights=df.weigh
        plot_df(df, order_coeff_dict)
```



Gewichten Sie dann auch noch den Punkt bei Eingabe -1 höher.

```
In [8]: weights = get_weights(df.x, [-.5, 0, .5], 100)
        reference = 'order=3'
        coeff_dict = {reference : order_coeff_dict[reference]}
        coeff_dict['weight_at_0'] = polyfit(df.x, df.y, order=3, weights=weights)
        plot_df(df, coeff_dict)
```



Was ändert sich?

Das Modell geht nun auch (zumindest beinah) durch den Ursprung.

Welche Form wird das optimale Modell haben und warum?

Ein optimales Modell gibt es nicht. Ein Polynom 8. Grades geht durch alle Datenpunkte, würde die Fehlerfunktion also minimieren, generalisiert aber schlecht.
Z.B. das Polynom 3. Grades stellt einen guten Kompromiss aus Underfitting und Overfitting dar.

3 Task 3.3 Gewichteter Fehler: multiple lokale Modelle:

Nehmen Sie nun an, Sie generieren die Gewichte r_n aus einer Gaussfunktion mit einer Varianz β^2 (z.B. $\beta^2 = 1$) und Mittelwert μ.

Welchen Effekt hat das auf die Modellierung?

Der Einfluss der Datenpunkte auf die Modellparameter fällt mit dem Abstand zum Mittelwert ab.
Es ist, als würde das Modell die Daten in der "Nähe" besser sehen, als die weit entfernten Samples.

Überlegen Sie dazu, welche Datenpunkte jeweils wichtig sind, wenn $\mu = -\frac{1}{2}, 0, \frac{1}{2}$ ist.

Wichtig sind die Datenpunke, die nahe an dem jeweiligen Mittelpunkt liegen.

$\beta^2 = 1$ scheint kein guter Wert zu sein.
Ich nehme $\beta = 0.3$ und verschiebe zusätzlich die ersten drei Datenpunkte um 0.2 nach unten.

```
In [9]: df = load_dataset() # reset
```

Diese Überlegungen legen nahe, das Datenmodell als Summe mehrerer lokaler Modelle zu wählen. Wie könnten diese lokalen Modelle aussehen, wenn wir Gaussgewichtungen mit $\mu_1 = -\frac{1}{2}$ und $\mu_2 = \frac{1}{2}$ und das Gesamtmodell als Summe von zwei lokalen Modellen wählen?

Es müssten Parabeln genügen, da jedes Modell nur einen Teil abbilden muss und sich der Datensatz recht gut
als zwei aneinander gelegte Parabeln beschreiben lässt

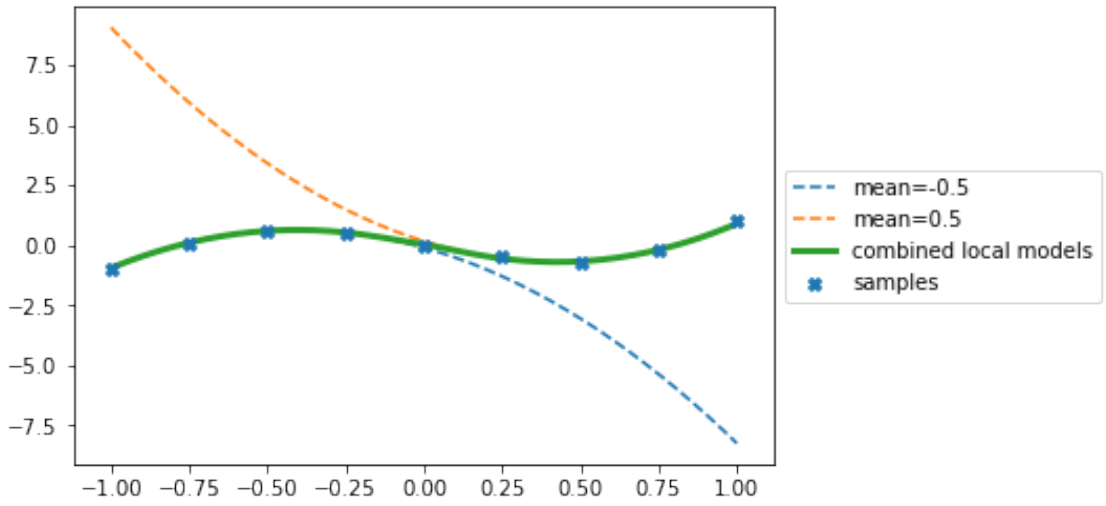
```
In [10]: import matplotlib.pyplot as plt
         from collections import namedtuple
         from functools import partial

In [11]: LocalModel = namedtuple('LocalModel', 'coeffs,distribution'.split(','))

         # evaluate the prediction of the model ensemble at the point x
         def combine_local_models(models, x):
             # calc weights at x (proportional to each pdf and summing to 1)
             weights = np.array([m.distribution.pdf(x) for m in models])
             weights /=  weights.sum()
             results = np.array([get_y(model.coeffs, np.array(x)) for model in models])
             return (results.ravel() * weights.ravel()).sum()

         # plot model enseble along with data samples (x,y)
         def plot_local_models(x, y, models, combination_only=False, **kwargs): # kwargs are pas
             combine_models = np.vectorize(partial(combine_local_models, models)) # to be called
             xmin, xmax = min(x), max(x)
             n_samples = 200
             ax = plt.gca()
             if not combination_only:
                 for model in models:
                     label = 'mean={}'.format(model.distribution.mean())
                     plot_weights(model.coeffs, label, xmin, xmax, ax, ls='--')
             xs = np.linspace(xmin, xmax, n_samples)
             ys = combine_models(xs)
             ax.plot(xs, ys, label='combined local models', lw=3, **kwargs)
             ax.scatter(x, y, label='samples', marker='X', zorder=10)
             plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))

In [12]: order = 2
         sigma = 0.2
         means = [-1/2, 1/2]
         dists = [norm(mean, sigma) for mean in means]
         coeffs = [polyfit(df.x, df.y, order, d.pdf(df.x)) for d in dists]
         models = [LocalModel(*tupel) for tupel in zip(coeffs, dists)]
         plot_local_models(df.x, df.y, models)
```



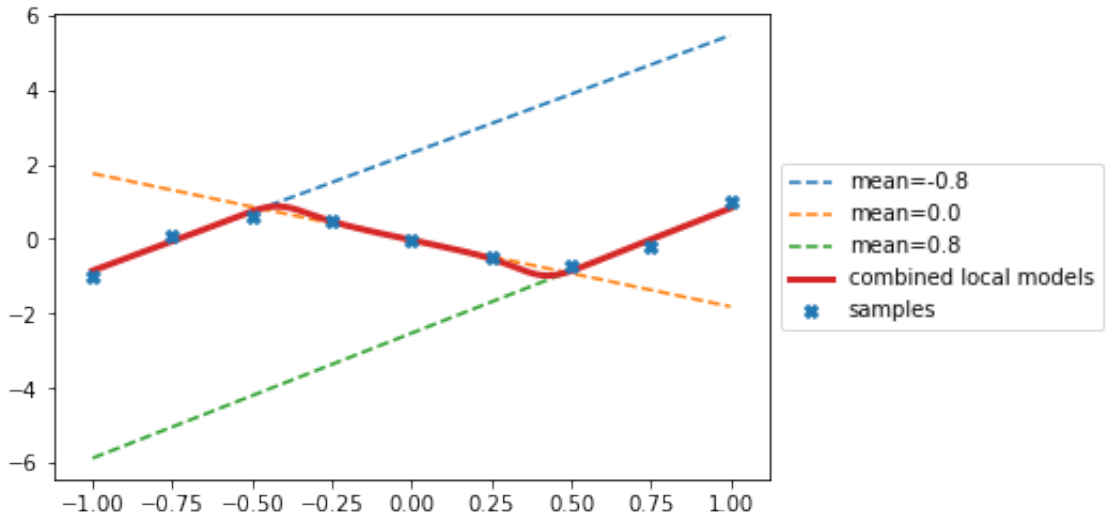
Wie, wenn man drei lokale Modelle verwenden wollte?

Dann könnte man es sogar mit Geraden versuchen:

Wo sollten dann die drei Mittelpunkte μ_1, μ_2, μ_3 für die jeweiligen Gaussfunktionen liegen, mit denen die Gewichte erzeugt werden, und was für ein Modell (d.h. welche Wahl der jeweiligen Polynomgrade) wäre sinnvoll?

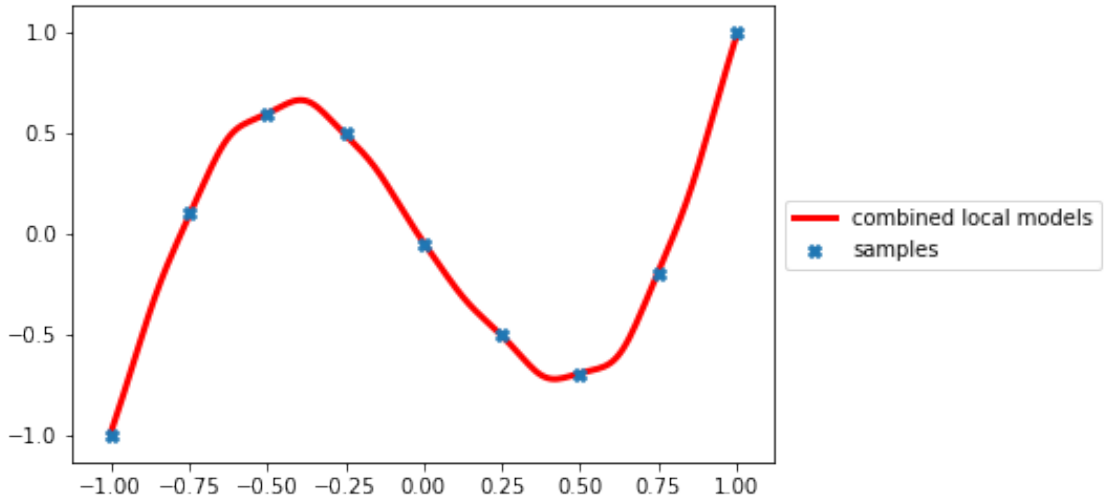
Der Datensatz lässt sich gut durch drei Graden darstellen, wobei die mittlere Gerade länger sein sollte, als die anderen.
Zum Beispiel könnten die Werte $-0.8, 0, 0.8$ als Mittelwerte verwendet werden.
Zusätzlich wähle ich σ noch kleiner, damit die geraden nicht von weit entfernten Punkten beeinflusst werden.

```
In [13]: order = 1
         sigma = 0.2
         means = [-0.8, 0, 0.8]
         dists = [norm(mean, sigma) for mean in means]
         coeffs = [polyfit(df.x, df.y, order, d.pdf(df.x)) for d in dists]
         models = [LocalModel(*tupel) for tupel in zip(coeffs, dists)]
         plot_local_models(df.x, df.y, models)
```



Extremfall: 9 Gewichte (je eines pro Datenpunkt), Grad 2 und kleines σ^2

```
In [14]: order = 1
         sigma = 0.1
         means = np.linspace(-1, 1, 9)
         dists = [norm(mean, sigma) for mean in means]
         coeffs = [polyfit(df.x, df.y, order, d.pdf(df.x)) for d in dists]
         models = [LocalModel(*tupel) for tupel in zip(coeffs, dists)]
         plot_local_models(df.x, df.y, models, combination_only=True, color='r')
```



4 Task 3.4 Gradientenberechnungen für tanh:

Zeige, dass gilt:

$$f(s) = \tanh(\beta s) \rightarrow f'(s) = \beta(1 - f(s)^2)$$

Ich betrachte zunächst $g(x) = \tanh(x)$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{6}$$
$$= \frac{e^{2x} - 1}{e^{2x} + 1} \frac{e^{-x}}{e^{-x}} \tag{7}$$
$$= \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{8}$$
$$\tag{9}$$

Ich wende die Quotientenregel an und erhalte:

$$\rightarrow \frac{\partial}{\partial x} tanh = \frac{\left(e^x + e^{-x}\right)\left(e^x + e^{-x}\right) - \left(e^x - e^{-x}\right)\left(e^x - e^{-x}\right)}{\left(e^x - e^{-x}\right)^2} \tag{10}$$
$$= 1 - \frac{\left(e^x - e^{-x}\right)\left(e^x - e^{-x}\right)}{\left(e^x - e^{-x}\right)^2} \tag{11}$$
$$= 1 - \tanh(x)^2 \tag{12}$$
$$\tag{13}$$

Dann wende ich die Kettenregel an und erhalte:

$$f'(s) = \tanh'(\beta s) \frac{\partial \beta x}{\partial x} \tag{14}$$
$$= \beta \left(1 - \tanh(\beta s)^2\right) \tag{15}$$
$$= \beta(1 - f(s)^2) \tag{16}$$
$$\tag{17}$$

Berechne dann den Gradienten der folgenden Fehler-funktion:

$$\nabla_{\omega_i} E(\vec{\omega}) = \nabla_{\omega_i} \frac{1}{2} \sum_{n=1}^{N} \left(t_n - \tanh\left(\sum_{d=1}^{D} \omega_d x_d \right) \right)^2$$

wobei wir einen einen D-dimensionalen Eingabevektor $\vec{x} \in \mathbf{R}^D$ und eindimensionale Ausgabe y annehmen.

Zunächst ziehe ich den Faktor $\frac{1}{2}$ in die Summe und erhalte:

$$\nabla_{\omega_i} E(\vec{\omega}) = \nabla_{\omega_i} \sum_{n=1}^{N} \frac{1}{2} \left(t_n - \tanh\left(\sum_{d=1}^{D} \omega_d x_d \right) \right)^2$$

Nun lässt sich $E(\vec{\omega})$ als Verkettung mehrerer Funktionen schreiben.
Da die Gradienten bzgl. der nächst tieferen Funktion trivial sind und später gebraucht werden, schreibe ich sie direkt daneben:

$$E(\vec{\omega}) = \sum_{n=1}^{N} E_n(\vec{\omega}) \qquad \rightarrow \frac{\partial E(\vec{\omega})}{\partial E_n(\vec{\omega})} = 1 \tag{18}$$

$$E_n(\vec{\omega}) = \frac{1}{2}\left(t_n - y_n\right)^2 \qquad \rightarrow \frac{\partial E_n(\vec{\omega})}{y_n} = -\left(t_n - y_n\right) = y_n - t_n \tag{19}$$

$$y_n = \tanh(z_n) \qquad \rightarrow \frac{\partial y_n}{\partial z_n} = z_n \tanh\left(1 - z_n^2\right) \tag{20}$$

$$z_n = \sum_{d=1}^{D} \omega_d x_{n,d} \qquad \rightarrow \frac{\partial z_n}{\partial \omega_i} = x_{n,i} \tag{21}$$

$$\tag{22}$$

$E(\vec{\omega})$ lässt sich also schreiben als $E\left(E_n\left(y_n\left(z_n\left(w_i\right)\right)\right)\right)$

$$\frac{\partial E(\vec{\omega})}{\partial \omega_i} = \sum_{n_1}^{N} \frac{\partial E_n(\vec{\omega})}{\partial \omega_i}$$

Nun lässt sich die Kettenregel anwenden, die wie folgt definiert ist:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial E_n(\vec{\omega})}{\partial \omega_i} = \frac{\partial E_n(\vec{\omega})}{\partial y_n}\frac{\partial y_n}{\partial z_n}\frac{\partial z_n}{\partial w_i} \tag{23}$$

$$= \left(y_n - t_n\right) z_n \tanh\left(z - t_n^2\right) x_{n,i} \tag{24}$$

$$\rightarrow \frac{\partial E(\vec{\omega})}{\partial \omega_i} = \sum_{n_1}^{N} \left(y_n - t_n\right) z_n \tanh\left(1 - z_n^2\right) x_{n,i} \tag{25}$$

$$= \sum_{n_1}^{N} \left(\left(\tanh\left(\sum_{d=1}^{D} \omega_d x_{n,d} \right) \right) - t_n \right) \left(\sum_{d=1}^{D} \omega_d x_{n,d} \right) \tanh\left(1 - \left(\sum_{d=1}^{D} \omega_d x_{n,d} \right)^2 \right) x_{n,i} \tag{26}$$

$$\tag{27}$$