



ISP AUFGABE 3

Gruppe 3, Team 5

06.06.2017

Dimitri Meier, Saeed Shanidar, Andreas Berks

dimitri.meier@haw-hamburg.de,
saeed.shanidar@haw-hamburg.de,
andreas.berks@haw-hamburg.de

Inhaltsverzeichnis

0. Allgemeine Definitionen.....	2
0.1. Konkatenation	2
0.2. Bias	3
1. Lineare Regression	4
1.1. Training mittels abgeschlossener Lösung.....	4
1.1.1. Definition der Trainingsdaten.....	4
1.1.2. Training des Modells	5
1.1.3. Aufstellen der Hypothese.....	7
1.1.4. Ergebnis	8
1.1.5. Beantwortung der Fragen	9
1.2. Training mittels Stochastic Gradient Descent (SGD)	10
1.2.1. Definition der Trainingsdaten.....	10
1.2.2. Training des Modells	11
1.2.3. Aufstellen der Hypothese.....	14
1.2.4. Ergebnis	15
1.2.5. Beantwortung der Fragen	17
1.3. Vorhersage von Immobilienpreisen mit linearer Regression.....	21
2. Logistische Regression.....	23
2.1. Training mittels SGD	23
2.1.1. Definition der Trainingsdaten.....	23
2.1.2. Training des Modells	23
2.1.3. Aufstellen der Hypothese.....	26
2.1.4. Ergebnis	27
2.1.5. Beantwortung der Fragen	29

0. Allgemeine Definitionen

0.1. Konkatenation

Wir definieren eine Funktion *Konkatenation* wie folgt:

$$\textit{Konkatenation} : (\mathbb{R}^{m,1})^n \rightarrow \mathbb{R}^{m,n}, x \mapsto y$$

$$\forall i \in [1, m], j \in [1, n] : y_{i,j} = (x_i)_j$$

Beispiel:

Seien zwei Vektoren x_1 und x_2 gegeben:

$$x_1 = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \quad x_2 = \begin{pmatrix} d \\ e \\ f \end{pmatrix}$$

Damit ergibt sich:

$$\textit{Konkatenation}(\{x_1, x_2\}) = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

Implementation in Python:

```
def concatenate(M1, M2):  
    return np.concatenate([M1, M2], axis = horizontal_axis)
```

0.2. Bias

Wir definieren eine Funktion *Bias* wie folgt:

$$Bias : \mathbb{R}^{m,n} \rightarrow \mathbb{R}^{m,1}, x \mapsto y$$

$$\forall i \in [1, m] : y_{i,1} = 1$$

Beispiel:

Sei eine Matrix X gegeben:

$$X = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

Damit ergibt sich:

$$Bias(X) = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Implementation in Python:

Wir definieren zuerst eine Hilfsfunktion, welche und die Anzahl der Zeilen und die Anzahl der Spalten der Matrix M liefert:

```
def get_dimensions(M):  
    return np.shape(M)
```

Nun definieren wir eine Funktion zum Erzeugen des Bias:

```
def create_bias(M):  
    m, n = get_dimensions(M)  
    return np.ones((m,1))
```

Weiterhin definieren wir eine Funktion zum Hinzufügen eines Bias:

```
def add_bias(M):  
    bias = create_bias(M)  
    return concatenate(bias, M)
```

1. Lineare Regression

1.1. Training mittels abgeschlossener Lösung

1.1.1. Definition der Trainingsdaten

Gegeben sind die folgenden Trainingsdaten:

x_1	y
0.86	2.49
0.09	0.83
-0.85	-0.25
0.87	3.1
-0.44	0.87
-0.43	0.02
-1.1	-0.12
0.4	1.81
-0.96	-0.83
0.17	0.43

$$x_1 = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix} \quad y = \begin{pmatrix} 2.49 \\ 0.83 \\ -0.25 \\ 3.1 \\ 0.87 \\ 0.02 \\ -0.12 \\ 1.81 \\ -0.83 \\ 0.43 \end{pmatrix}$$

Wir definieren nun eine Matrix X welche sämtliche Trainingsdaten beinhaltet.
Dazu konkatenieren wir sämtliche Spalten der Trainingsdaten:

$$Spalten = \{x_1\}$$

$$X = \text{Konkatenation}(Spalten)$$

1.1.2. Training des Modells

Im Folgenden definieren nun eine Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten.

Zuerst fügen wir der Matrix X den Bias x_0 hinzu.

Die entstehende Matrix bezeichnen wir als X_{Bias} .

$$x_0 = Bias(X)$$

$$X_{Bias} = Konkatenation(\{x_0\} \cup Spalten)$$

Die entsprechenden Matrizen X und X_{Bias} stehen wie folgt aus:

$$X = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix} \quad X_{Bias} = \begin{pmatrix} 1.0 & 0.86 \\ 1.0 & 0.09 \\ 1.0 & -0.85 \\ 1.0 & 0.87 \\ 1.0 & -0.44 \\ 1.0 & -0.43 \\ 1.0 & -1.1 \\ 1.0 & 0.4 \\ 1.0 & -0.96 \\ 1.0 & 0.17 \end{pmatrix}$$

Nun bestimmen wir die Gewichte θ mittels abgeschlossener Lösung:

$$\theta = (X_{Bias}^T X_{Bias})^{-1} X_{Bias}^T y$$

In diesem Beispiel entstehen dadurch die folgenden Gewichte:

$$\theta = \begin{pmatrix} 1.05881340999 \\ 1.61016841718 \end{pmatrix}$$

Implementation in Python:

Wir definieren zuerst Hilfsfunktionen, um Matrizen zu multiplizieren, zu transponieren und zu invertieren:

```
def multiply(M1, M2):  
    return np.dot(M1, M2)  
  
def transpose(M):  
    return np.transpose(M)  
  
def inverse(M):  
    return np.linalg.inv(M)
```

Nun definieren wir eine Funktion zum Bestimmen der Gewichte:

```
def calcThetaClosedForm(X, y):  
    XT = transpose(X)  
    result = multiply(XT, X)  
    result = inverse(result)  
    result = multiply(result, XT)  
    result = multiply(result, y)  
    return result
```

Wir speichern uns die Werte von θ und nutzen diese in der Definition von $predict(X_{Test})$.

Damit ist die Definition der Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten abgeschlossen.

1.1.3. Aufstellen der Hypothese

Weiterhin definieren wir eine Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable.

Es gilt für X_{Test} :

$$X_{Test} = \text{Konkatenation}(\{x_{Test1}\})$$

$$Spalten_{Test} = \{x_{Test1}\}$$

Zuerst fügen wir der Matrix X_{Test} den Bias x_{0Test} hinzu.

Die entstehende Matrix bezeichnen wir als $X_{TestBias}$.

$$x_{Test0} = \text{Bias}(X_{Test})$$

$$X_{TestBias} = \text{Konkatenation}(\{X_{TestBias}\} \cup Spalten_{Test})$$

Nun berechnen wir die Hypothese $h(X_{TestBias})$:

$$h(X_{TestBias}) = X_{TestBias} \theta$$

$$h(X_{TestBias})_{i,k} = \sum_{j=0}^n X_{TestBias i,j} \cdot \theta_{j,k}$$

Implementation in Python:

```
def calcHypothesis(theta, X):  
    thetaT = transpose(theta)  
    result = multiply(X, thetaT)  
    return result
```

Das Ergebnis der Hypothese $h(X_{TestBias})$ geben wir als Ergebnis zurück.

Damit ist die Definition der Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable abgeschlossen.

1.1.4. Ergebnis

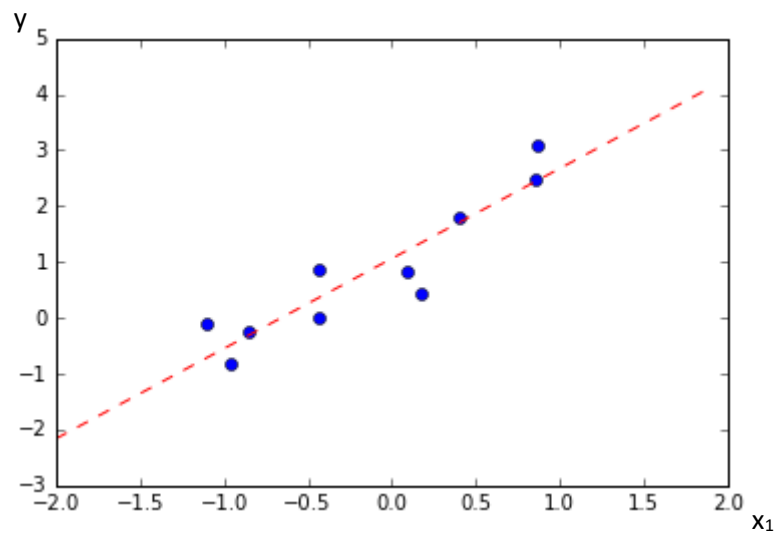
Zur Erinnerung: Gegeben waren die folgenden Trainingsdaten:

x_1	y
0.86	2.49
0.09	0.83
-0.85	-0.25
0.87	3.1
-0.44	0.87
-0.43	0.02
-1.1	-0.12
0.4	1.81
-0.96	-0.83
0.17	0.43

$$x_1 = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix}$$

$$y = \begin{pmatrix} 2.49 \\ 0.83 \\ -0.25 \\ 3.1 \\ 0.87 \\ 0.02 \\ -0.12 \\ 1.81 \\ -0.83 \\ 0.43 \end{pmatrix}$$

Das Ergebnis der Hypothese sieht wie folgt aus:



1.1.5. Beantwortung der Fragen

1) Was ist der naheliegende Nachteil an der abgeschlossenen Lösung, sobald ein besonders großer Datensatz verwendet werden soll?

In diesem Fall ist die abgeschlossene Lösung sehr speicheraufwendig und in der Regel nur selten anwendbar.

Beispiel:

Anzahl der Samples = 1.000.000

Anzahl der Features = 100.000

Für die Matrix $M = X^T X$ enthält in diesem Fall $100.000 * 100.000 = 10^{10}$ Elemente. Sind diese Elemente vom Datentyp `float` (64 Bit) wird etwa 80 Gigabyte Hauptspeicher benötigt.

1.2. Training mittels Stochastic Gradient Descent (SGD)

1.2.1. Definition der Trainingsdaten

Gegeben sind die folgenden Trainingsdaten:

x_1	y
0.86	2.49
0.09	0.83
-0.85	-0.25
0.87	3.1
-0.44	0.87
-0.43	0.02
-1.1	-0.12
0.4	1.81
-0.96	-0.83
0.17	0.43

$$x_1 = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix} \quad y = \begin{pmatrix} 2.49 \\ 0.83 \\ -0.25 \\ 3.1 \\ 0.87 \\ 0.02 \\ -0.12 \\ 1.81 \\ -0.83 \\ 0.43 \end{pmatrix}$$

Wir definieren nun eine Matrix X welche sämtliche Trainingsdaten beinhaltet.
Dazu konkatenieren wir sämtliche Spalten der Trainingsdaten:

$$Spalten = \{x_1\}$$

$$X = \text{Konkatenation}(Spalten)$$

1.2.2. Training des Modells

Im Folgenden definieren nun eine Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten.

Zuerst fügen wir der Matrix X den Bias x_0 hinzu.

Die entstehende Matrix bezeichnen wir als X_{Bias} .

$$x_0 = Bias(X)$$

$$X_{Bias} = Konkatenation(\{x_0\} \cup Spalten)$$

Die entsprechenden Matrizen X und X_{Bias} stehen wie folgt aus:

$$X = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix} \quad X_{Bias} = \begin{pmatrix} 1.0 & 0.86 \\ 1.0 & 0.09 \\ 1.0 & -0.85 \\ 1.0 & 0.87 \\ 1.0 & -0.44 \\ 1.0 & -0.43 \\ 1.0 & -1.1 \\ 1.0 & 0.4 \\ 1.0 & -0.96 \\ 1.0 & 0.17 \end{pmatrix}$$

Nun bestimmen wir die Gewichte θ mittels Stochastic Gradient Descent:

Zuerst initialisieren wir eine Menge für *costs* zu speichernden Werte der Kosten und initialisieren zusätzlich die Gewichte θ mit dem Wert 1.0:

$$costs = \{\}$$
$$\theta = \left\{ \begin{pmatrix} 1.0 \\ 1.0 \\ \dots \\ 1.0 \\ 1.0 \end{pmatrix} \right\} \text{ n Elemente}$$

Anschließend werden nach und nach die Gewichte in einer Schleife angenähert:

```
for iteration in [1, iterations]:  
    for i in [1, m]:  
        for j in [1, n]:  
             $\theta_j = \theta_j + \alpha (y_i - h(x_i)) x_{i,j}$   
         $costs = costs \cup J(\theta)$ 
```

Als Kostenfunktion nutzen wir die sum-of-squares-errors-Funktion:

$$J(\theta) = \frac{1}{2} \sum_m^{i=1} (h(x_i) - y_i)^2$$

Seien nun die folgenden Parameter für das Training festgelegt:

$alpha = 0.001, iterations = 1000$

In diesem Beispiel entstehen dadurch die folgenden Gewichte:

$$\theta = \begin{pmatrix} 1.05350947347 \\ 1.60561505101 \end{pmatrix}$$

Implementation in Python:

Wir definieren zuerst Hilfsfunktionen, um die Gewichte zu initialisieren:

```
def initialize_theta(X):  
    m, n = get_dimensions(X)  
    return np.ones((n))
```

Nun definieren wir eine Funktion zum Bestimmen der Gewichte:

```
# Initialisierung der Gewichte (inklusive Bias-Term)  
# self.weights in der Form [n_features + 1]  
self.weights = initialize_theta(X)  
  
# Array zum Speichern des Errors für jeden SGD-Schritt  
self.cost = []  
  
# SGD Schleife über Iterationen  
for _ in range(iterations):  
    # Schleife über Datensätze  
    for i in range(m):  
  
        # Update der Gewichte  
        # Schleife über Parameter (n+1 durch Bias-Term)  
        for j in range(n+1):  
            temp = y[i] - calc_hypothesis(self.weights, X[i])  
            self.weights[j] = self.weights[j] + alpha * temp * X[i][j]  
  
        # Berechne SSE des SGD-Schritts mit den aktualisierten Gewichten  
self.cost = self.cost + [calc_sum_of_squared_errors(self.weights, X, y)]
```

Dabei ist die Kostenfunktion wie folgt definiert:

```
def calc_sum_of_squared_errors(w, X, y):  
    m, n = get_dimensions(X)  
    result = 0  
    for i in range(1, m):  
        result += (y[i] - calc_hypothesis(w, X[i])) ** 2  
    result = result / 2  
    return result
```

Wir speichern uns die Werte von θ und nutzen diese in der Definition von $predict(X_{Test})$.

Damit ist die Definition der Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten abgeschlossen.

1.2.3. Aufstellen der Hypothese

Weiterhin definieren wir eine Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable.

Es gilt für X_{Test} :

$$X_{Test} = \text{Konkatenation}(\{x_{Test1}\})$$

$$Spalten_{Test} = \{x_{Test1}\}$$

Zuerst fügen wir der Matrix X_{Test} den Bias x_{0Test} hinzu.

Die entstehende Matrix bezeichnen wir als $X_{TestBias}$.

$$x_{Test0} = \text{Bias}(X_{Test})$$

$$X_{TestBias} = \text{Konkatenation}(\{X_{TestBias}\} \cup Spalten_{Test})$$

Nun berechnen wir die Hypothese $h(X_{TestBias})$:

$$h(X_{TestBias}) = X_{TestBias} \theta$$

$$h(X_{TestBias})_{i,k} = \sum_{j=0}^n X_{TestBias i,j} \cdot \theta_{j,k}$$

Implementation in Python:

```
def calcHypothesis(theta, X):  
    thetaT = transpose(theta)  
    result = multiply(X, thetaT)  
    return result
```

Das Ergebnis der Hypothese $h(X_{TestBias})$ geben wir als Ergebnis zurück.

Damit ist die Definition der Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable abgeschlossen.

1.2.4. Ergebnis

Zur Erinnerung: Gegeben waren die folgenden Trainingsdaten:

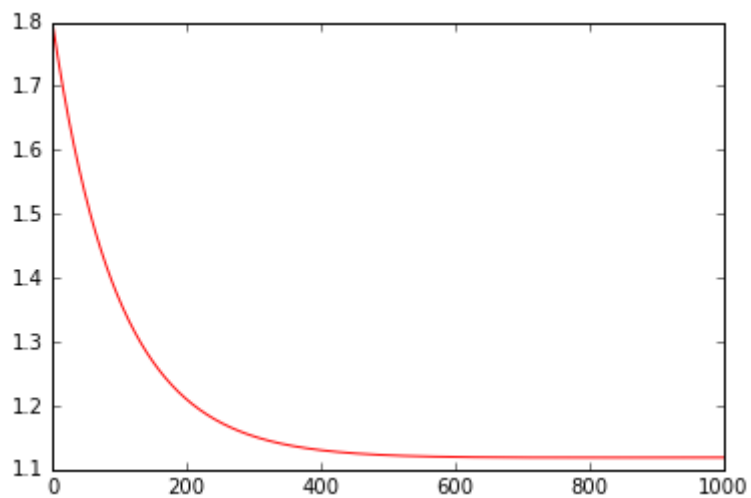
x_1	y
0.86	2.49
0.09	0.83
-0.85	-0.25
0.87	3.1
-0.44	0.87
-0.43	0.02
-1.1	-0.12
0.4	1.81
-0.96	-0.83
0.17	0.43

$$x_1 = \begin{pmatrix} 0.86 \\ 0.09 \\ -0.85 \\ 0.87 \\ -0.44 \\ -0.43 \\ -1.1 \\ 0.4 \\ -0.96 \\ 0.17 \end{pmatrix} \quad y = \begin{pmatrix} 2.49 \\ 0.83 \\ -0.25 \\ 3.1 \\ 0.87 \\ 0.02 \\ -0.12 \\ 1.81 \\ -0.83 \\ 0.43 \end{pmatrix}$$

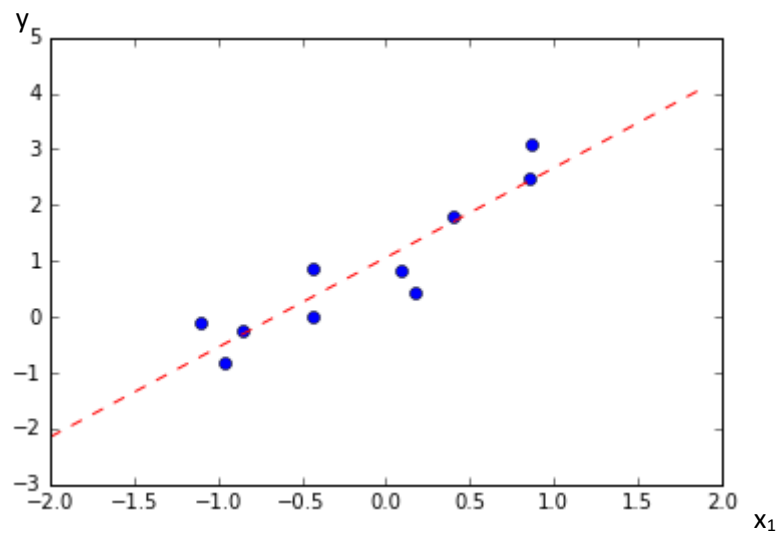
Seien nun die folgenden Parameter für das Training festgelegt:

$$\alpha = 0.001, \text{iterations} = 1000$$

Die Fehlerkurve sieht wie folgt aus:



Das Ergebnis der Hypothese sieht wie folgt aus:



1.2.5. Beantwortung der Fragen

1) Probieren sie unterschiedliche Werte für die Parameter alpha und iterations in der predict()-Funktion aus und betrachten Sie die Auswirkung auf den Graphen des CostWertes.

An den folgenden Testdaten lässt sich folgendes ableiten:

- Je kleiner der Wert für alpha, desto genauer, jedoch auch langsamer die Annäherung
- Je größer der Wert für iterations, desto genauer die Annäherung, jedoch wird irgendwann Konvergenz erreicht

Ergebnis der geschlossenen Form:

$$\theta = \begin{pmatrix} 1.05881340999 \\ 1.61016841718 \end{pmatrix}$$

Auswirkungen auf das Ergebnis der Gewichte:

$\alpha = 0.1, \text{iterations} = 10$

$$\theta = \begin{pmatrix} 1.00765915784 \\ 1.62453957299 \end{pmatrix}$$

$\alpha = 0.1, \text{iterations} = 100$

$$\theta = \begin{pmatrix} 1.00973007855 \\ 1.63122340317 \end{pmatrix}$$

$\alpha = 0.1, \text{iterations} = 1000$

$$\theta = \begin{pmatrix} 1.00973007855 \\ 1.63122340317 \end{pmatrix}$$

$\alpha = 0.1, \text{iterations} = 10000$

$$\theta = \begin{pmatrix} 1.00973007855 \\ 1.63122340317 \end{pmatrix}$$

$\alpha = 0.01, \text{iterations} = 10$

$$\theta = \begin{pmatrix} 0.992487384982 \\ 1.23107298601 \end{pmatrix}$$

$\alpha = 0.01, \text{iterations} = 100$

$$\theta = \begin{pmatrix} 1.05350947347 \\ 1.60561505101 \end{pmatrix}$$

$\alpha = 0.01, \text{iterations} = 1000$

$$\theta = \begin{pmatrix} 1.05510432059 \\ 1.61172165337 \end{pmatrix}$$

$\alpha = 0.01, \text{iterations} = 10000$

$$\theta = \begin{pmatrix} 1.05510432059 \\ 1.61172165337 \end{pmatrix}$$

$\alpha = 0.001, \text{iterations} = 10$

$$\theta = \begin{pmatrix} 0.997658304558 \\ 1.02852627072 \end{pmatrix}$$

$\alpha = 0.001, \text{iterations} = 100$

$$\theta = \begin{pmatrix} 0.99579621399 \\ 1.23041284218 \end{pmatrix}$$

$\alpha = 0.001, \text{iterations} = 1000$

$$\theta = \begin{pmatrix} 1.05689685516 \\ 1.60423833205 \end{pmatrix}$$

$\alpha = 0.001, \text{iterations} = 10000$

$$\theta = \begin{pmatrix} 1.05845358299 \\ 1.61031794299 \end{pmatrix}$$

$\alpha = 0.0001, \text{iterations} = 10$

$$\theta = \begin{pmatrix} 0.999742675707 \\ 1.00291674727 \end{pmatrix}$$

$\alpha = 0.0001, \text{iterations} = 100$

$$\theta = \begin{pmatrix} 0.997713030451 \\ 1.02851629376 \end{pmatrix}$$

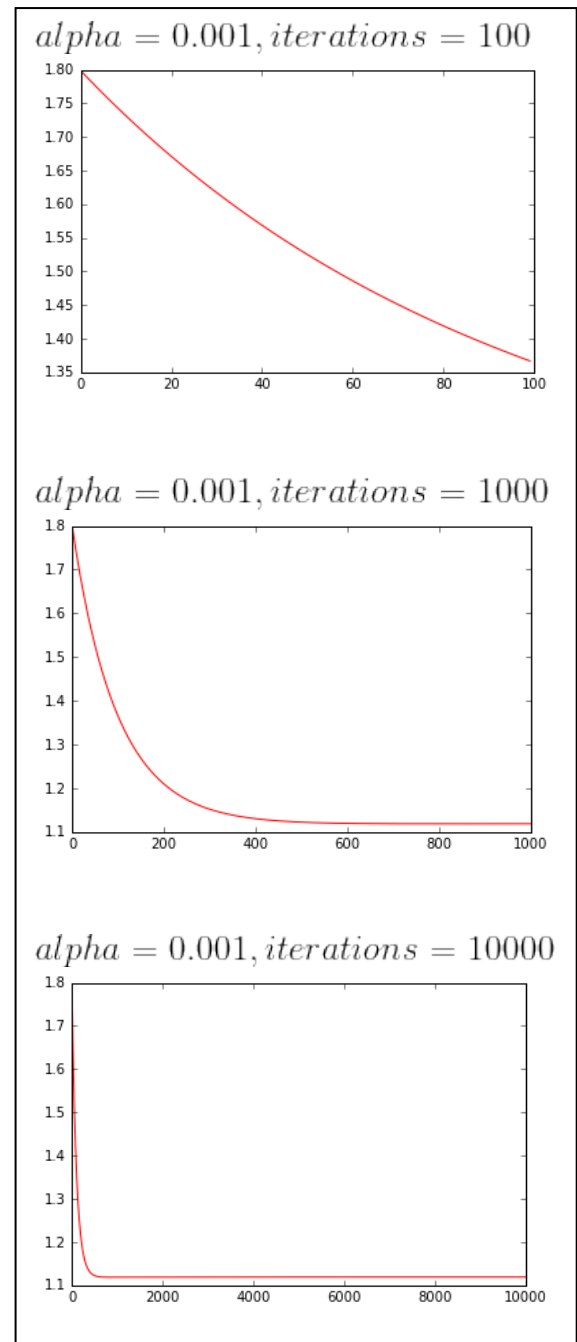
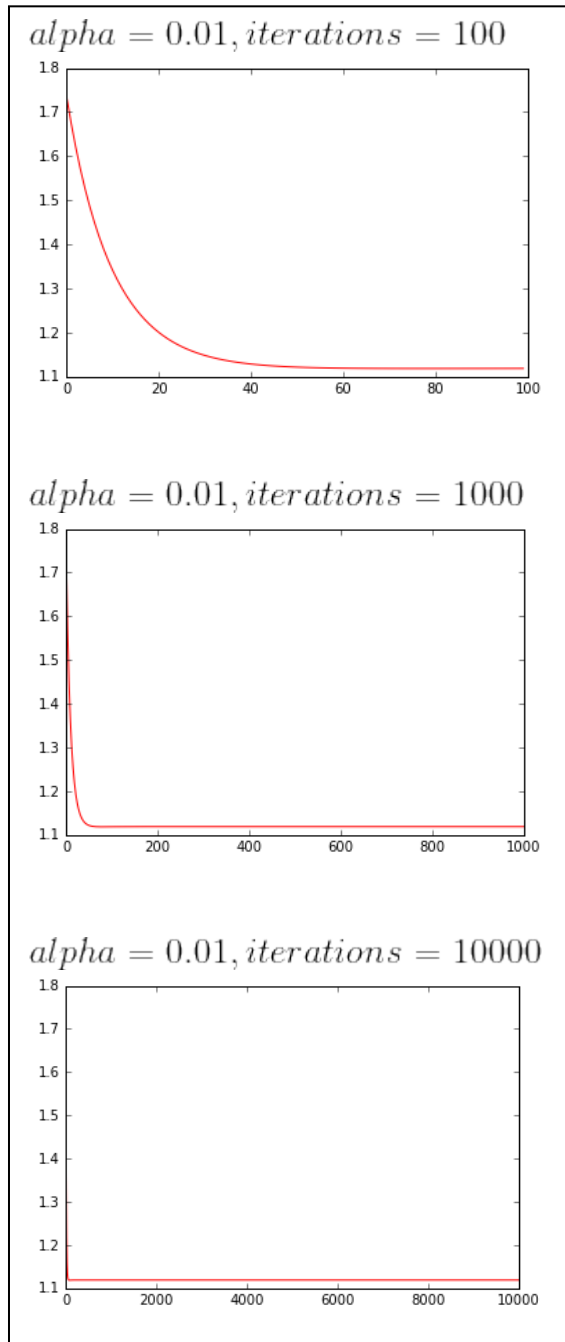
$\alpha = 0.0001, \text{iterations} = 1000$

$$\theta = \begin{pmatrix} 0.996117326835 \\ 1.23034720164 \end{pmatrix}$$

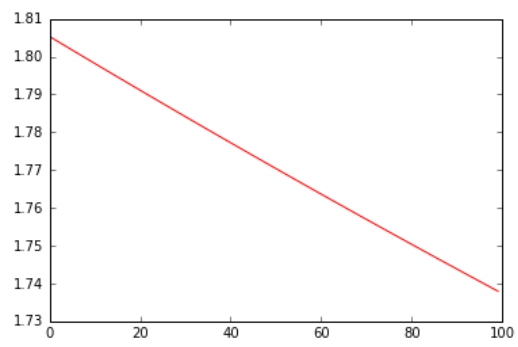
$\alpha = 0.0001, \text{iterations} = 10000$

$$\theta = \begin{pmatrix} 1.05722454986 \\ 1.60410615124 \end{pmatrix}$$

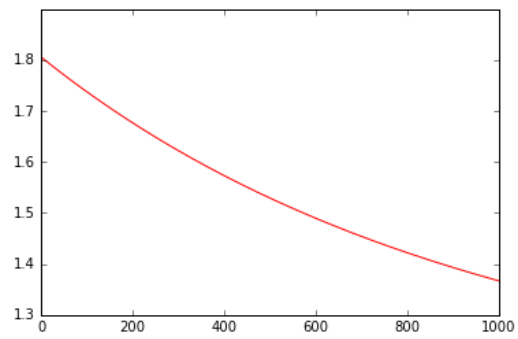
Auswirkungen auf den Graphen des CostWertes:



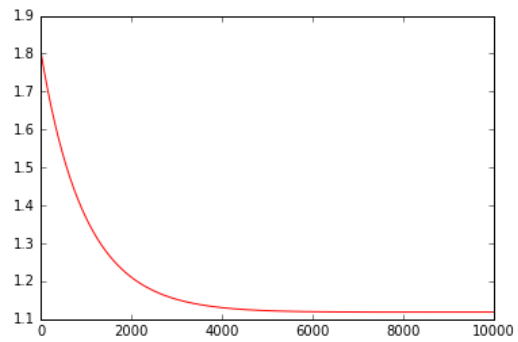
$\alpha = 0.0001, iterations = 100$



$\alpha = 0.0001, iterations = 1000$



$\alpha = 0.0001, iterations = 10000$



2) Warum wird in der Praxis Stochastic Gradient Descent (SGD) gegenüber Batch Gradient Descent (s. Vorlesung) bevorzugt?

Batch Gradient Descent geht über das gesamte Trainingsset um die Gewichte zu aktualisieren. Dadurch besteht die Gefahr, dass der Hauptspeicher „gesprengt“ wird. Dies führt zu einer langsameren Ausführung (durch Nutzung einer Auslagerungsdatei).

Außerdem übertritt Stochastic Gradient Descent im Gegensatz zu Batch Gradient Descent lokale Optima durch „noisy“ Gradient. Das bedeutet, dass beim Stochastic Gradient Descent ein „Rauschen“ während des Training ausgeglichen wird.

1.3. Vorhersage von Immobilienpreisen mit linearer Regression

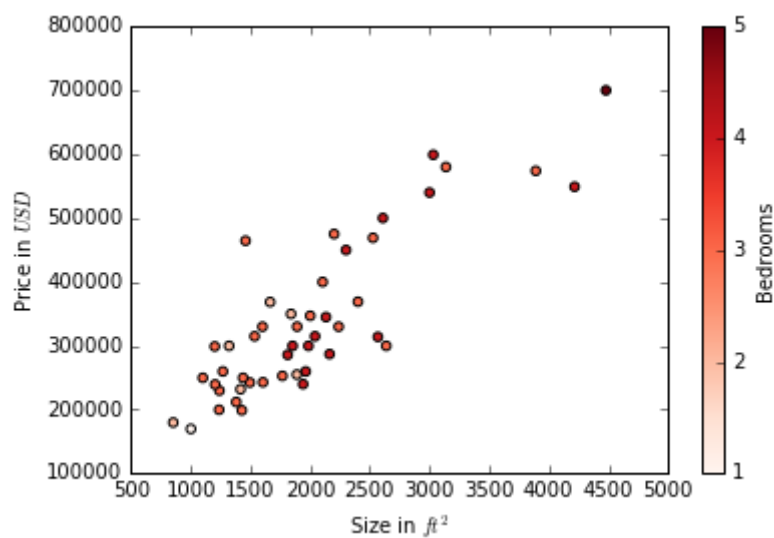
Wir nutzen folgenden Formel zur Standardisierung:

$$x'_i = \frac{x_i - \mu}{\sigma}$$

Implementation in Python:

```
def standardize(data):  
    average = np.average(data)  
    standard_diviation = np.std(data)  
  
    for x in range(len(data)):  
        data[x] = np.abs(data[x] - average) / standard_diviation
```

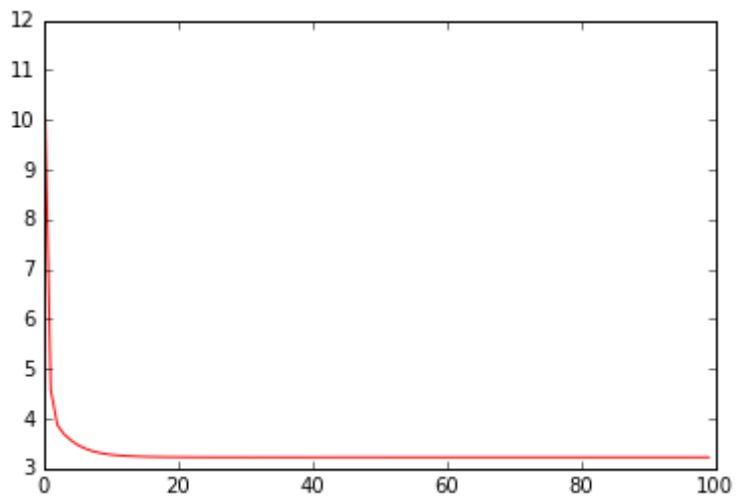
Gegeben waren die folgenden Trainingsdaten:



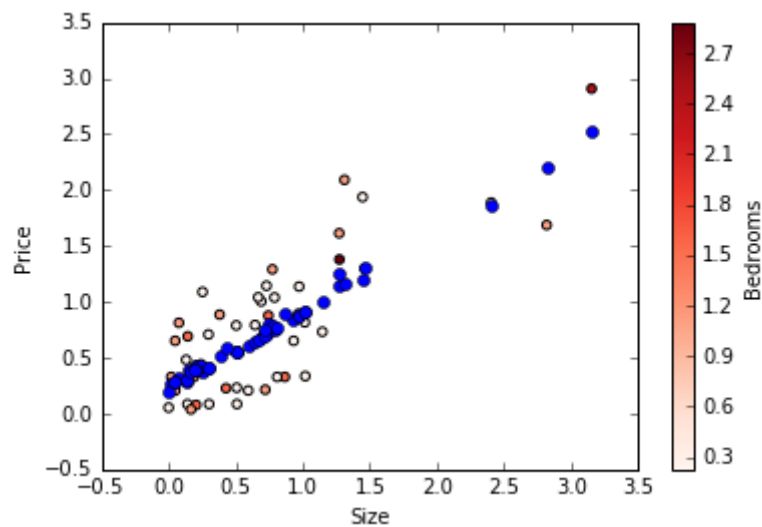
Seien nun die folgenden Parameter für das Training festgelegt:

$\alpha = 0.001, \text{iterations} = 1000$

Die Fehlerkurve sieht wie folgt aus:



Das Ergebnis der Hypothese sieht wie folgt aus:



2. Logistische Regression

2.1. Training mittels SGD

2.1.1. Definition der Trainingsdaten

Wir definieren eine Matrix X welche sämtliche Trainingsdaten beinhaltet.
Dazu konkatenieren wir sämtliche Spalten der Trainingsdaten:

$$Spalten = \{x_1, x_2\}$$

$$X = Konkatenation(Spalten)$$

2.1.2. Training des Modells

Im Folgenden definieren nun eine Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten.

Zuerst fügen wir der Matrix X den Bias x_0 hinzu.

Die entstehende Matrix bezeichnen wir als X_{Bias} .

$$x_0 = Bias(X)$$

$$X_{Bias} = Konkatenation(\{x_0\} \cup Spalten)$$

Nun bestimmen wir die Gewichte θ mittels Stochastic Gradient Descent:

Zuerst initialisieren wir eine Menge für *costs* zu speichernden Werte der Kosten und initialisieren zusätzlich die Gewichte θ mit dem Wert 1.0:

$$costs = \{\}$$

$$\theta = \left\{ \begin{pmatrix} 1.0 \\ 1.0 \\ \dots \\ 1.0 \\ 1.0 \end{pmatrix} \right\} \text{ n Elemente}$$

Anschließend werden nach und nach die Gewichte in einer Schleife angenähert:

```
for iteration in [1, iterations]:  
    for i in [1, m]:  
        for j in [1, n]:  
             $\theta_j = \theta_j + \alpha (y_i - h(x_i)) x_{i,j}$   
         $costs = costs \cup J(\theta)$ 
```


Als Kostenfunktion nutzen wir die Log-Likelihood-Funktion:

$$J(\theta) = \left[\sum_{i=1}^m y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i)) \right] + \frac{\lambda}{2} \sum_n \theta_j^2$$

Die Hypothese definieren wir mit Hilfe der Sigmoid-Funktion $g(z)$:

$$h(X) = g(\theta^T X) = \frac{1}{1 + e^{-\theta^T X}}$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Seien nun die folgenden Parameter für das Training festgelegt:

$$\alpha = 0.001, \text{iterations} = 1000$$

In diesem Beispiel entstehen dadurch die folgenden Gewichte:

$$\theta = \begin{pmatrix} -0.0162526192467 \\ -0.230444070701 \\ 2.89348779659 \end{pmatrix}$$

Implementation in Python:

Wir definieren zuerst Hilfsfunktionen, um die Gewichte zu initialisieren:

```
def initialize_theta(X):  
    m, n = get_dimensions(X)  
    return np.ones((n))
```

Nun definieren wir eine Funktion zum Bestimmen der Gewichte:

```
# Initialisierung der Gewichte (inklusive Bias-Term)  
# self.weights in der Form [n_features + 1]  
self.weights = initialize_theta(X)  
  
# Array zum Speichern des Errors für jeden SGD-Schritt  
self.cost = []  
  
# SGD Schleife über Iterationen  
for _ in range(iterations):  
    # Schleife über Datensätze  
    for i in range(m):  
  
        # Update der Gewichte  
        # Schleife über Parameter (n+1 durch Bias-Term)  
        for j in range(n+1):  
            temp = y[i] - h_x(self.weights, X[i])  
            self.weights[j] = self.weights[j] + alpha * temp * X[i][j]  
  
        # Berechne SSE des SGD-Schritts mit den aktualisierten Gewichten  
self.cost = self.cost + [cost_fn(self.weights, X, y)]
```

Dabei ist die Hypothese wie folgt definiert:

```
# Sigmoid Funktion  
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))  
  
# Hypothese h(x)  
def h_x(w, X):  
    return sigmoid(multiply(X, transpose(w)))
```

Dabei ist die Kostenfunktion wie folgt definiert:

```
def cost_fn(w, X, y):  
    _lambda = 10  
    m, n = np.shape(X)  
  
    l = 0  
    for i in range(1, m):  
        term1 = y[i] * np.log(h_x(w, X[i]))  
        term2 = (1 - y[i]) * np.log(1 - h_x(w, X[i]))  
        l += (term1 + term2)  
  
    result = 0  
    for j in range(1, n):  
        result += (w[j] ** 2)  
    result = (_lambda / 2) * result  
  
    result += l  
    return result
```

Wir speichern uns die Werte von θ und nutzen diese in der Definition von $predict(X_{Test})$.

Damit ist die Definition der Methode $fit(X, y)$ zum Training des Modells anhand von gegebenen Trainingsdaten abgeschlossen.

2.1.3. Aufstellen der Hypothese

Weiterhin definieren wir eine Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable.

Es gilt für X_{Test} :

$$X_{Test} = \text{Konkatenation}(\{x_{Test1}\})$$

$$Spalten_{Test} = \{x_{Test1}\}$$

Zuerst fügen wir der Matrix X_{Test} den Bias x_{0Test} hinzu.

Die entstehende Matrix bezeichnen wir als $X_{TestBias}$.

$$x_{Test0} = \text{Bias}(X_{Test})$$

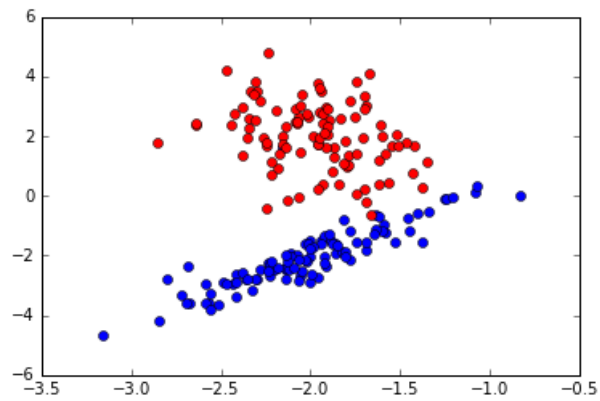
$$X_{TestBias} = \text{Konkatenation}(\{X_{TestBias}\} \cup Spalten_{Test})$$

Das Ergebnis der Hypothese $h(X_{TestBias})$ geben wir als Ergebnis zurück.

Damit ist die Definition der Methode $predict(X_{Test})$ zur numerischen Vorhersage der Zielvariable abgeschlossen.

2.1.4. Ergebnis

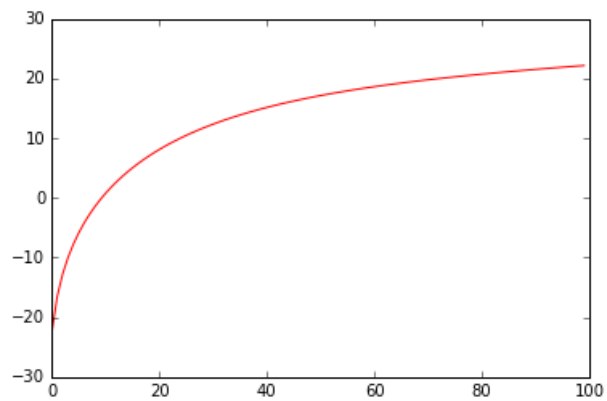
Gegeben waren die folgenden Trainingsdaten:



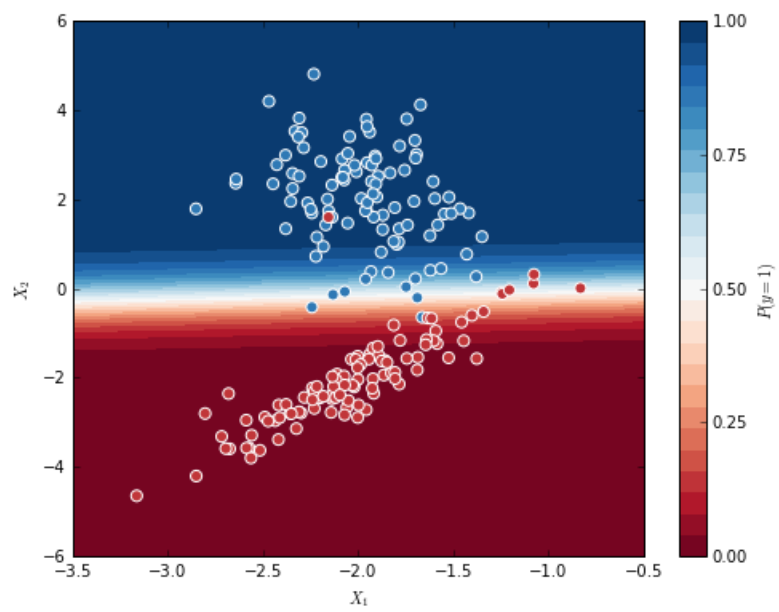
Seien nun die folgenden Parameter für das Training festgelegt:

$\alpha = 0.001, iterations = 1000$

Die Fehlerkurve sieht wie folgt aus:



Das Ergebnis der Hypothese sieht wie folgt aus:



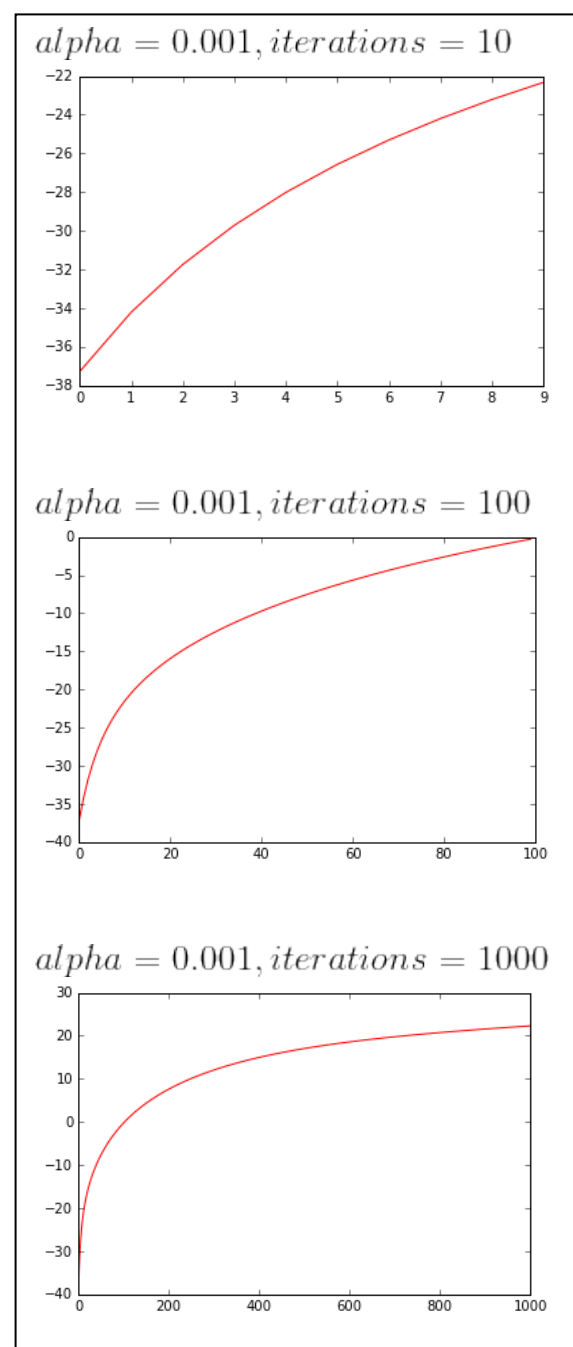
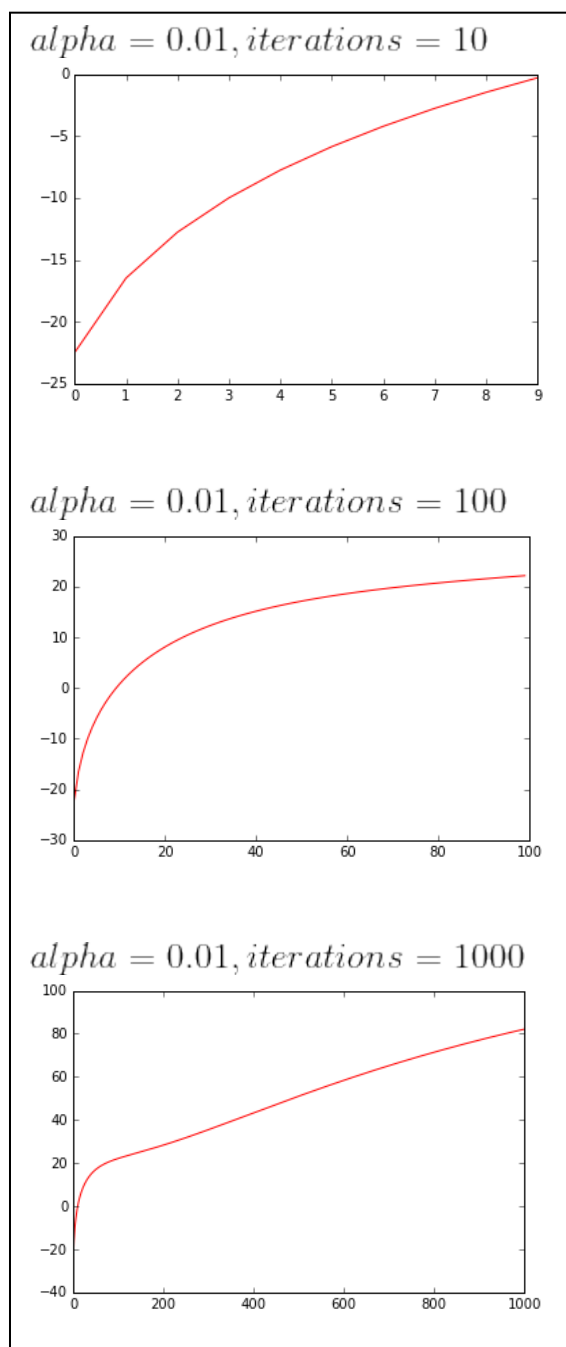
2.1.5. Beantwortung der Fragen

1) Überprüfen Sie, welche Auswirkung die Veränderung des Wertes iterations und alpha in der Funktion fit() auf die Qualität der Decision-Boundary haben und dokumentieren Sie ihre Ergebnisse im Protokoll.

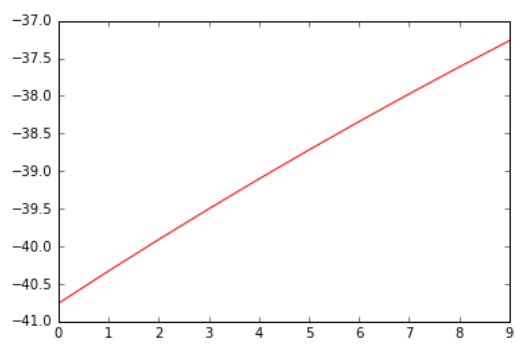
An den folgenden Testdaten lässt sich folgendes ableiten:

- Je kleiner der Wert für alpha, desto genauer, jedoch auch langsamer die Annäherung
- Je größer der Wert für iterations, desto genauer die Annäherung, jedoch wird irgendwann Konvergenz erreicht

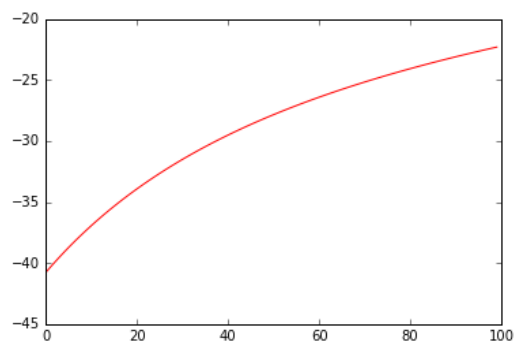
Auswirkungen auf den Graphen des Cost Wertes:



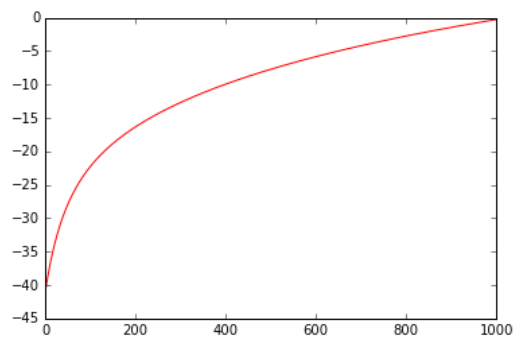
$\alpha = 0.0001, iterations = 10$



$\alpha = 0.0001, iterations = 100$



$\alpha = 0.0001, iterations = 1000$



2) Was müssten Sie tun, damit Sie mehr als zwei Klassen vorhersagen können?

Wir definieren für jede Klasse eigene Vektor mit Zielvariablen y .

Wenn für eigene gegebene Klasse das Sample der Trainingsdaten X_i mit $i \in [1, m]$ zu der entsprechenden Klasse gehört, setzen wir den Wert für $y_i = 1$ (true).

Wenn nicht, setzen wir den Wert für $y_i = 0$ (false).

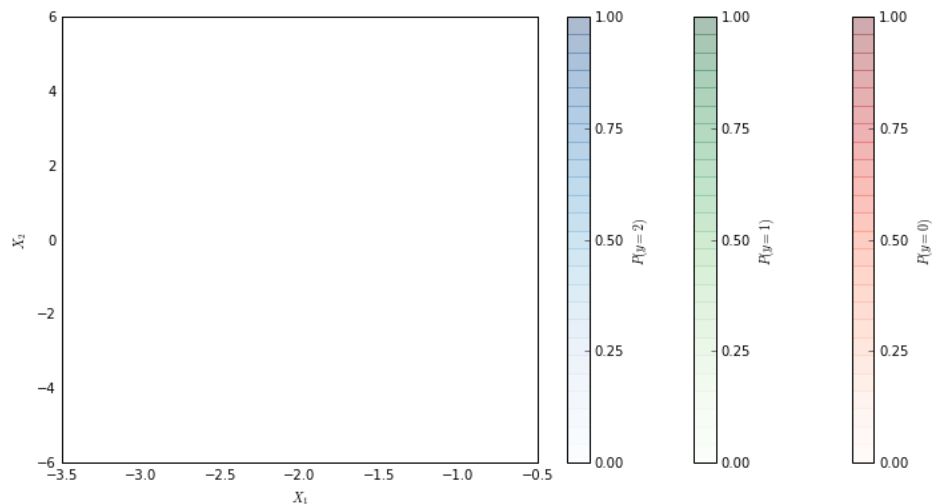
Wir führen nun für jede Klasse ein Training durch, um den entsprechenden Wahrscheinlichkeitsgraphen zu erhalten.

Wir zeichnen nun jeden dieser Wahrscheinlichkeitsgraphen mit einem Alpha-Wert von $\frac{1}{\text{Anzahl der Klassen}}$

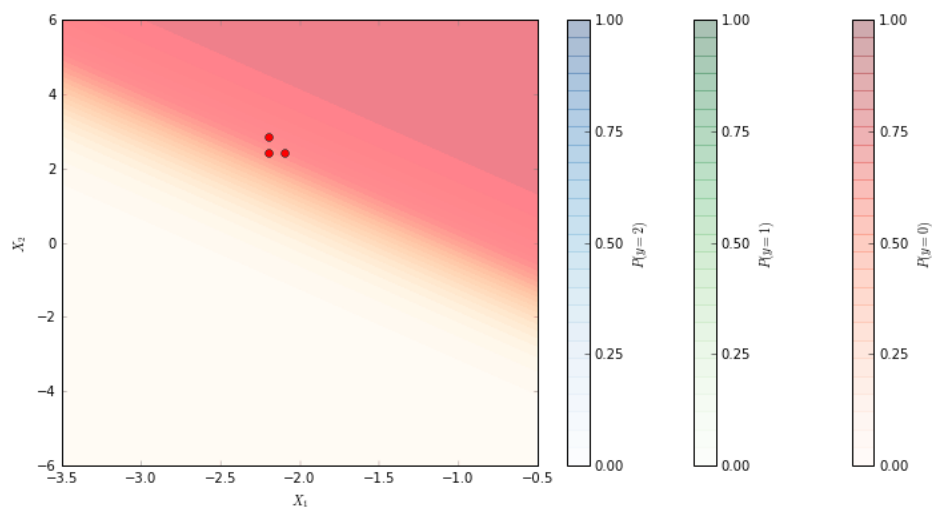
nach und nach übereinander.

Beispiel:

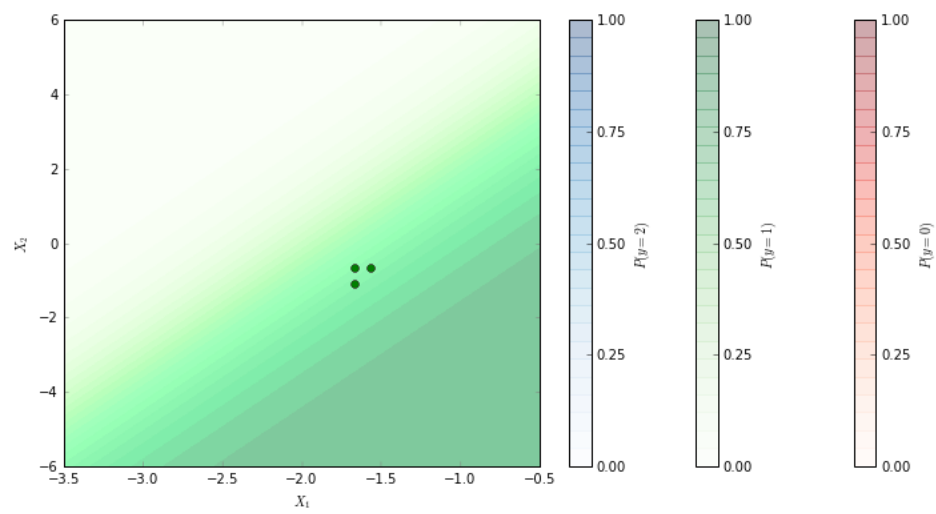
Darstellung des Wahrscheinlichkeitsgraphen keiner Klasse:



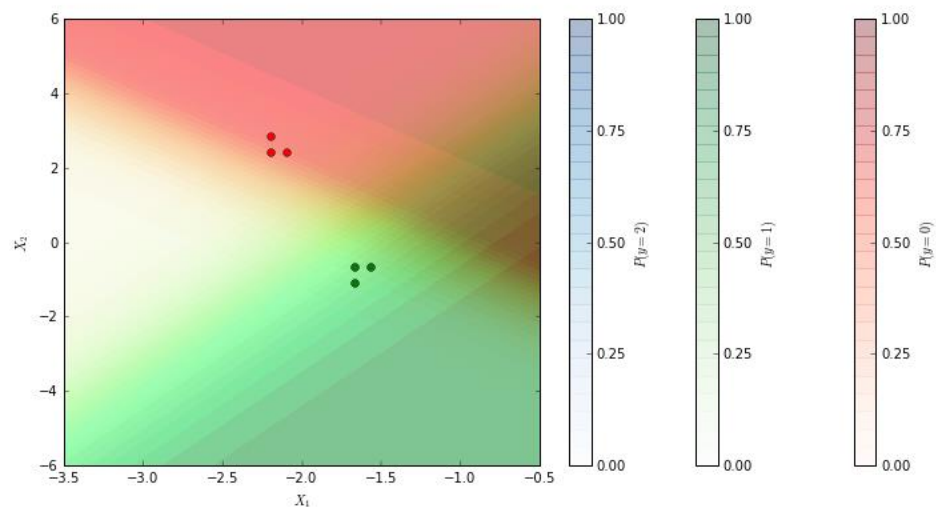
Darstellung des Wahrscheinlichkeitsgraphen der Klasse 0:



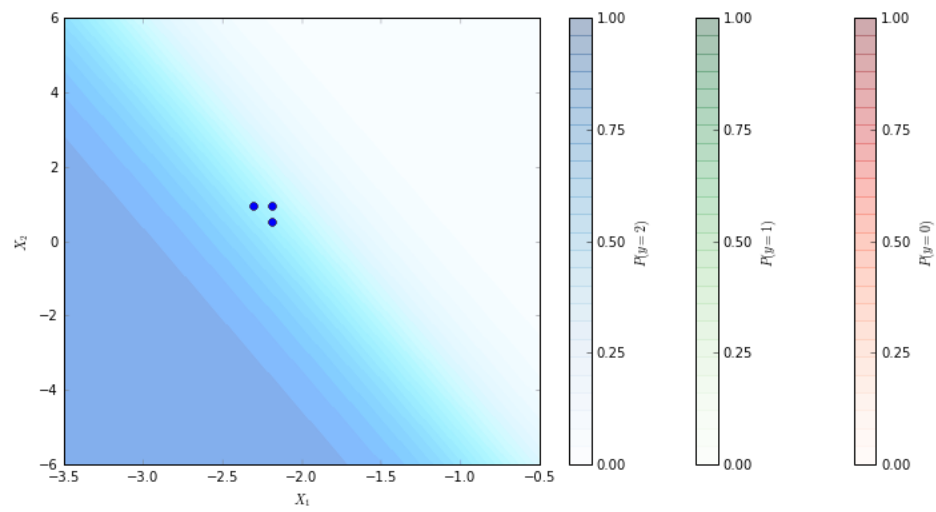
Darstellung des Wahrscheinlichkeitsgraphen der Klasse 1:



Darstellung des Wahrscheinlichkeitsgraphen der Klassen 0 und 1:



Darstellung des Wahrscheinlichkeitsgraphen der Klasse 3:



Darstellung des Wahrscheinlichkeitsgraphen der Klassen 1, 2 und 3:

