My sample book

The Jupyter Book Community

Jul 14, 2021

CONTENTS

This is a small sample book to give you a feel for how book content is structured.

:::{note} Here is a note! :::

And here is a code block:

```
e = mc^2
```

Check out the content pages bundled with this sample book to see more.

I wonder if we can have some math formulae as well:

$\alpha = 3$

or like this:

```
$\alpha = 3$
```

and even the more advanced stuff?

```
import matplotlib.pyplot as plt
%matplotlib inline
```

CHAPTER

ONE

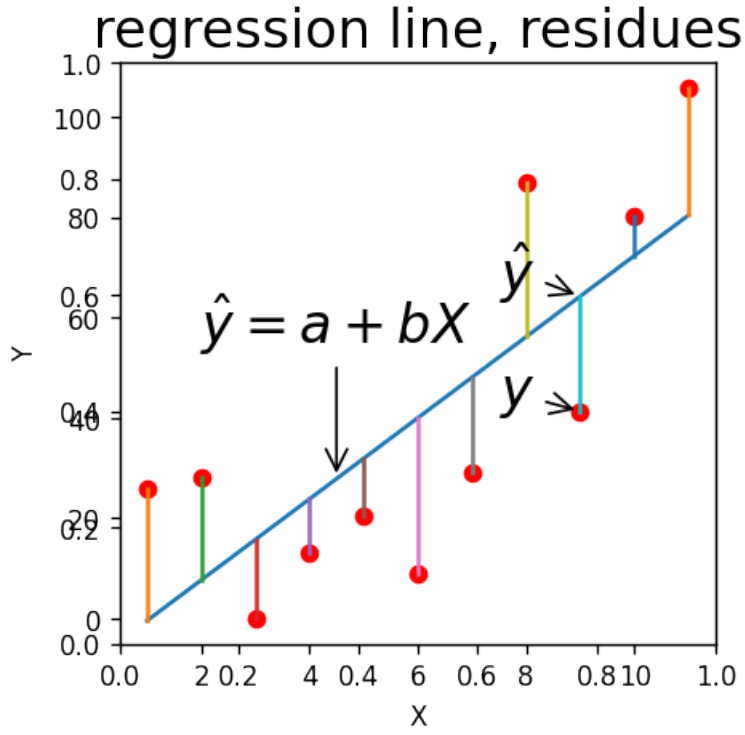
LINEARE REGRESSION

In der nachfolgenden Zelle werden zuerst Daten geladen, die zur Veranschaulichung der linearen Regression dienen. Anschliessend wird ein lineares Modell mit Hilfe der der Klasse Lineare Regression aus `sklearn.linear_model` gerechnet. Die Vorhersage (d.h. die Geradengleichung) ergibt sich aus den Koeffizienten durch $y = a + bX$.

```
from sklearn.linear_model import LinearRegression
import numpy as np
import matplotlib.pyplot as plt
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')
model = LinearRegression()
model.fit(X, y)
y_hat = model.coef_ * X + model.intercept_
```

Warum wird für **X** immer ein Grossbuchstabe verwendet und für **y** ein kleiner Buchstabe ?

Die Matrix der Variablen X wird gross geschrieben, da in Matrix-Notation Matrizen immer mit grossen Buchstaben bezeichnet werden, Vektoren - so wie die abhängige Variable y - werden mit kleinen Buchstaben benannt.



Der Plot zeigt die berechnete Regressionsgerade, sowie die Abweichungen (die Fehler) der wirklichen Messwerte von dieser Geraden. Diese Abweichungen werden als **Residuen** bezeichnet, weil es der Anteil der gemessenen Werte ist, der "übrig bleibt", d.h. nicht durch das Modell erklärt werden kann. Vorhergesagte Variablen werden meist mit einem Dach (Hut) bezeichnet, sowie $\hat{y}$.

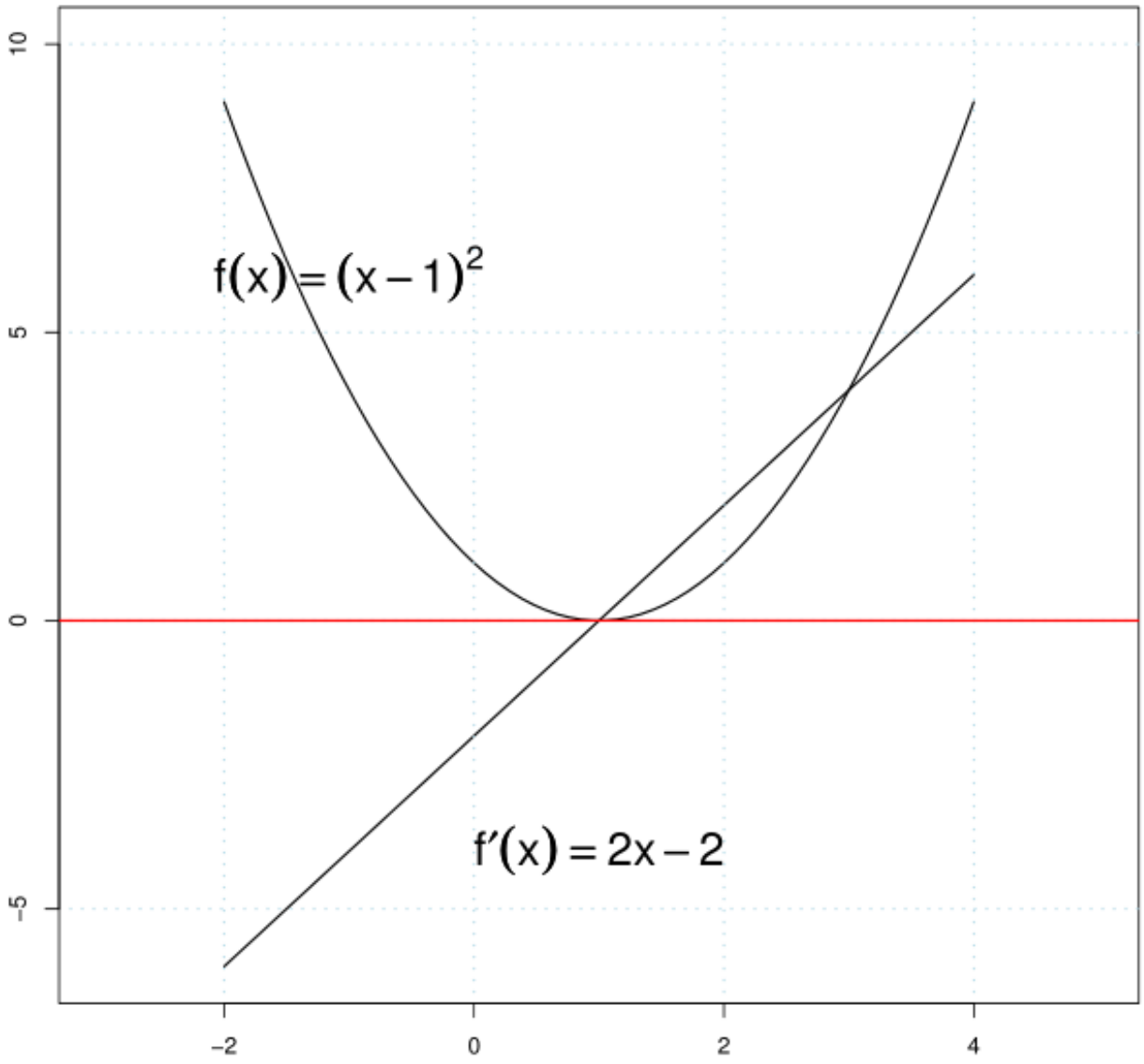
CHAPTER

TWO

ANALYTISCHE HERLEITUNG DER PARAMETER DER LINEAREN REGRESSION

Allgemein kann man den Nullpunkt einer quadratischen Funktion bestimmen, indem man ihre erste Ableitung gleich 0 setzt. Die erste Ableitung gibt die Steigung der Funktion an. In der Physik ist dies of die Beschleunigung. Die Steigung ist am Minimum der Funktion schliesslich 0. Man beachte, dass quadratische Funktionen immer nur einen Maximalwert haben können.

Nachfolgend ist dieser Sachverhalt für die quadratische Funktion $f(x) = (x-1)^2$ dargestellt. Die Ableitung $2x - 2$ ist ebenfalls eingetragen. Bei dem Minimum der Funktion ist die erste Ableitung gleich 0 (die Stelle an der der Funktionsgraph, der der ersten Ableitung und die rote, horizontale Linie sich schneiden).



Die Parameter einer linearen Regression können analytisch berechnet werden. Dazu wird der quadrierte Fehler $(y_i - \hat{y}_i)^2$ über alle Messwerte aufsummiert. Diese Summe wird nach den Parametern abgeleitet und gleich 0 gesetzt. Somit erhält man die Stelle an der die quadratische Funktion keine Steigung (erste Ableitung ist Steigung) hat. Weil eine quadratische Funktion als einzige Nullstelle der Steigung ein Minimum hat, erhalten wir somit die Parameter an dem Minimum unserer quadratischen Fehlerfunktion.

2.1 derivative of the error term $(y - \hat{y})^2$:

- für $\hat{y}$ können wir auch schreiben: $a + b \cdot x$, dies ist die Vorhersage mit Hilfe der Regression-Gerade (der Geraden-Gleichung):

$$\sum_i^n (y_i - \hat{y_i})^2 = \sum_i^n [y_i - (a + b \cdot x_i)]^2$$

- wir leiten diese Fehler-Funktion nach a ab und setzen diese erste Ableitung gleich 0 (Hierbei wird die Kettenregel verwendet):

$$\begin{aligned} \frac{\delta \sum_i^n (y_i - \hat{y_i})^2}{\delta a} &= -2 \sum_i^n y_i + 2b \sum_i^n x_i + 2na = 0 \\ 2na &= 2 \sum_i^n y_i - 2b \sum_i^n x_i \\ a &= \frac{2 \sum_i^n y_i}{2n} - \frac{2b \sum_i^n x_i}{2n} \end{aligned}$$

- die Summe über alle x_i geteilt durch n – die Anzahl aller Beobachtungen – ergibt den Mittelwert $\bar{x}$, gleiches gilt für $\bar{y}$:

$$a = \bar{y} - b\bar{x}$$

- die Lösung für b ergibt sich analog; hier ersetzen wir a mit obigen Ergebnis und erhalten:

$$b = \frac{\frac{1}{n} \sum_i^n (x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{n} \sum_i^n (x_i - \bar{x})^2} = \frac{\text{cov}_{xy}}{\text{var}_x}$$

- Vereinfacht ist die Former: Kovarianz der beiden Variablen x und y geteilt durch die Varianz von x.

Nachfolgend wird demonstriert, wie die hergeleiteten Formeln, in python angewendet dieselben Parameter-Schätzer ergeben wie die aus der Klasse `LineareRegression` aus `sklearn.linear_model`. Dies soll einfach nur demonstrieren, dass die alles ganz leicht zu rechnen ist und keiner komplizierten Algorithmen bedarf.

```
# we can easily verify these results
print(f'the parameter b is the coefficient of the linear model {model.coef_}')
print(f'the parameter a is called the intercept of the model because it indicates\n␣
 ↪where the regression line intercepts the y-axis at x=0 {model.intercept_}')

cov_xy =(1/X.shape[0]) * np.dot((X - np.mean(X)).T,y - np.mean(y))[0][0]
var_x = (1/X.shape[0]) * np.dot((X - np.mean(X)).T,X - np.mean(X))[0][0]
b = cov_xy/var_x
a = np.mean(y)-b*np.mean(X)
print(f'\nour self-computed b parameter is: {b}')
print(f'our self-computed a parameter is: {a}')
```

```
the parameter b is the coefficient of the linear model [[8.07912445]]
the parameter a is called the intercept of the model because it indicates
 where the regression line intercepts the y-axis at x=0 [-8.49032154]

our self-computed b parameter is: 8.079124453577005
our self-computed a parameter is: -8.490321540681798
```

CHAPTER

THREE

MULTIVARIATE CASE: MORE THAN ONE X VARIABLE

Für Multivariate Lineare Regression kann die Schreibweise mit Matrizen zusammengefasst werden. Dafür kann es lohnend sein, sich die Matrizen-Multiplikation noch einmal kurz anzusehen.

$$
\begin{aligned}
y_1 &= a + b_1 \cdot x_{11} + b_2 \cdot x_{21} + \cdots + b_p \cdot x_{p1} \\
y_2 &= a + b_1 \cdot x_{12} + b_2 \cdot x_{22} + \cdots + b_p \cdot x_{p2} \\
&\cdots\cdots \\
y_i &= a + b_1 \cdot x_{1i} + b_2 \cdot x_{2i} + \cdots + b_p \cdot x_{pi}
\end{aligned}
$$

$$
\begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_i \end{bmatrix} = a + \begin{bmatrix} x_{11} & x_{21} & x_{31} & ... & x_{p1} \\ x_{12} & x_{22} & x_{32} & ... & x_{p2} \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ x_{1i} & x_{2i} & x_{3i} & ... & x_{pi} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_p \end{bmatrix}
$$

Den konstanten inercept Term (a) können wir mit in den Vektor der Parameter **b** aufnehmen, indem wir in **X** eine Einser-Spalte hinzufügen. Somit wird die Schreibweise sehr kompakt und der intercept a wird nicht mehr explizit aufgeführt:

$$
\begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_i \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{21} & x_{31} & ... & x_{p1} \\ 1 & x_{12} & x_{22} & x_{32} & ... & x_{p2} \\ & ... & ... & ... & ... & ... \\ & ... & ... & ... & ... & ... \\ 1 & x_{1i} & x_{2i} & x_{3i} & ... & x_{pi} \end{bmatrix} \cdot \begin{bmatrix} a \\ b_1 \\ b_2 \\ . \\ . \\ b_p \end{bmatrix}
$$

In Matrizen-Schreibweise können wir jetzt einfach schreiben: $\mathbf{y} = \mathbf{Xb}$

CHAPTER

FOUR

DERIVATION OF $\mathbf{B}$ FOR THE MATRIX NOTATION

Anschliessend wird die Berechnung der Parameter der Multivariaten Regression in Matrizen-Schreibweise erläutert. Konzeptionell ist dies nicht vom univariaten Fall verschieden. Diese Formel wird nur hergeleitet um demonstrieren zu können, wie das Ergebnis der expliziten Berechnung in Python mit dem aus der sklearn Klasse `LinearRegression` übereinstimmt.

- we expand the error term:

$$\begin{aligned}\min =&(\mathbf{y}-\hat{\mathbf{y}})^2 = (\mathbf{y}-\mathbf{Xb})'(\mathbf{y}-\mathbf{Xb}) = \\ &(\mathbf{y}'-\mathbf{b}'\mathbf{X}')(\mathbf{y}-\mathbf{Xb}) = \\ &\mathbf{y}'\mathbf{y}-\mathbf{b}'\mathbf{X}'\mathbf{y}-\mathbf{y}'\mathbf{Xb}+\mathbf{b}'\mathbf{X}'\mathbf{Xb} = \\ &\mathbf{y}'\mathbf{y}-2\mathbf{b}'\mathbf{X}'\mathbf{y}+\mathbf{b}'\mathbf{X}'\mathbf{Xb}\end{aligned}$$

- derivative of the error term with respect to **b**
- we set the result equal to zero and solve for **b**

$$\begin{aligned}\frac{\delta}{\delta\mathbf{b}} =& -2\mathbf{X}'\mathbf{y}+2\mathbf{X}'\mathbf{Xb}=0 \\ 2\mathbf{X}'\mathbf{Xb} =& 2\mathbf{X}'\mathbf{y} \\ \mathbf{b} =& (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}\end{aligned}$$

Hierbei bedarf es der Inversion des Kreuzproduktes der Variablen-Matrix $(\mathbf{X}'\mathbf{X})^{-1}$. Die Matrizen-Inversion ist für grosse Anzahl von Variablen mathematisch sehr aufwändig und kann unter Umständen zu Ungenauigkeiten führen. In der Vergangenheit wurde viel an Algorithmen geforscht um die Inversion schneller und stabiler zu machen. Oftmals stehen Fehlermeldungen in Zusammenhang mit diesem Berechnungsschritt.

CHAPTER

FIVE

POLYNOMIAL REGRESSION AS AN EXAMPLE FOR MORE THAN ONE VARIABLE

Um einfach Multivariate Lineare Regression an einem Beispiel zeigen zu können wird die quadratische Regression (ein Spezial-Fall der Multivariaten Regression) eingeführt. Eine neue Variable entsteht durch das Quadrieren der bisherigen univiaraten Variable x. Das Praktische ist, dass sich der Sachverhalt der Multivariaten Regression noch immer sehr schön 2-dimensional darstellen lässt. $y = a + b_1 x + b_2 x^2$

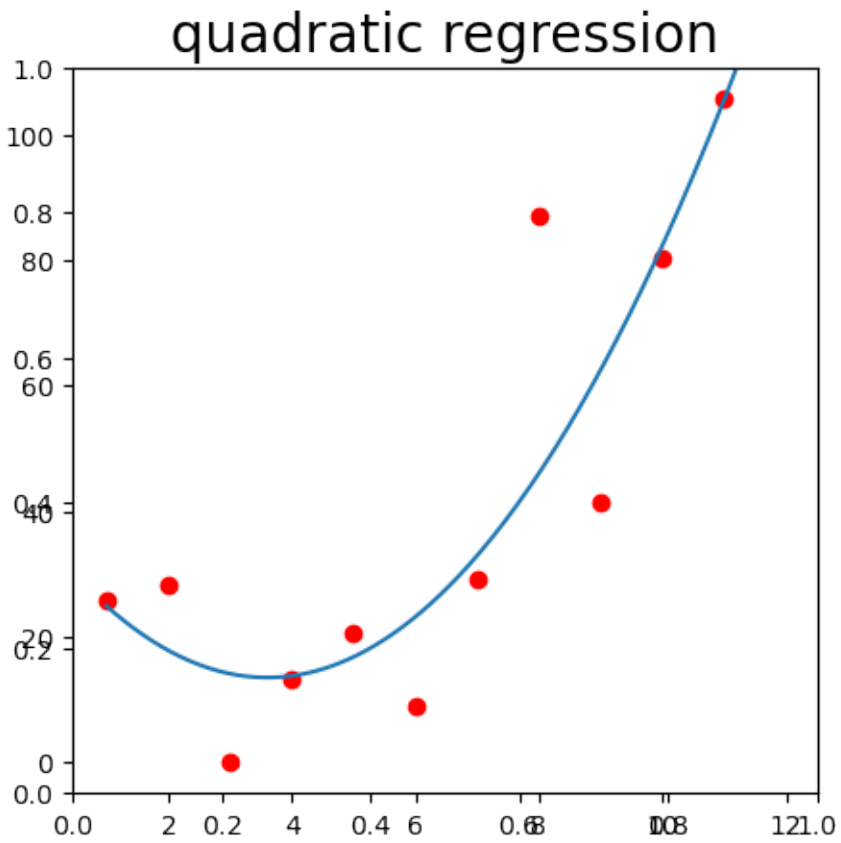
Hier ist zu beachten:

- wir haben jetzt zwei Variablen und können folglich unsere Formel in Matrizen-Schreibweise anwenden
- mehr Variablen führen hoffentlich zu einem besseren Modell
- durch den quadratischen Term ist die resultierende Regressions-Funktion keine Gerade mehr. **Der Ausdruck "linear" in Linearer Regression bedeutet dass die Funktion linear in den Parametern $a, \mathbf{b_1}, \mathbf{b_2}$ ist. Für alle Werte einer Variablen $\mathbf{x_1}$ gilt der gleiche Parameter $\mathbf{b_1}$. Es bedeutet nicht, dass die Regressions-Funktion durch eine gerade Linie gegeben ist!**

Nachfolgend fügen wir die weitere Variable durch Quadrieren der bisherigen Variable hinzu und berechnen abermals das Lineare Modell aus `sklearn.linear_model`.

```
from numpy.linalg import inv
# polynomial
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')
# underdetermined, ill-posed: infinitely many solutions
X = np.c_[X, X**2]

# the x (small x) is just for plotting purpose
x = np.arange(1, 12, 0.05).reshape((-1, 1))
x = np.c_[x, x**2]

model.fit(X, y)
y_hat = np.dot(x , model.coef_.T)  + model.intercept_
```

```
(-5.0, 110.77315979942053)
```



Jetzt berechnen wir die Parameter der Multiplen Linearen Regression mit Hilfe der hergeleiteten Formeln. Hierfür fügen wir zu den bisherigen Variablen x und x^2 noch eine Einser-Spalte für den intercpet ein. `np.dot` berechnet das dot-product zweier Variablen. Um das Kreuzprodukt von $\mathbf{X}$ berechnen zu können, muss eine der beiden Matrizen transponiert werden. Dies geschieht durch `.T`. `inv` invertiert das Kreuzprodukt.

`coefs = np.dot(np.dot(inv(np.dot(X_intercept.T,X_intercept)),X_intercept.T),y)` ist gleichbedeutend mit:

$$\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

```
# again we can compare the parameters of the model with those resulting from
# our derived equation:
# b=(X'X)^{-1} X'y
from numpy.linalg import inv

# first we have to add the intercept into our X-Variable; we rename it X_intercept
X_intercept = np.c_[np.ones(X.shape[0]), X]
coefs = np.dot(np.dot(inv(np.dot(X_intercept.T,X_intercept)),X_intercept.T),y)
print(f'the parameter b is the coefficient of the linear model {model.coef_}')
print(f'the parameter a is called the intercept of the model because it indicates\n␣
 ↪where the regression line intercepts the y-axis at x=0 {model.intercept_}')

print(f'our coefs already include the intercept: {coefs}')
```

```
the parameter b is the coefficient of the linear model [[-12.14930516   1.68570247]]
the parameter a is called the intercept of the model because it indicates
 where the regression line intercepts the y-axis at x=0 [35.33794262]
our coefs already include the intercept: [[ 35.33794262]
```

(continues on next page)

(continued from previous page)

```
 [-12.14930516]
 [  1.68570247]]
```

5.1 Overfitting

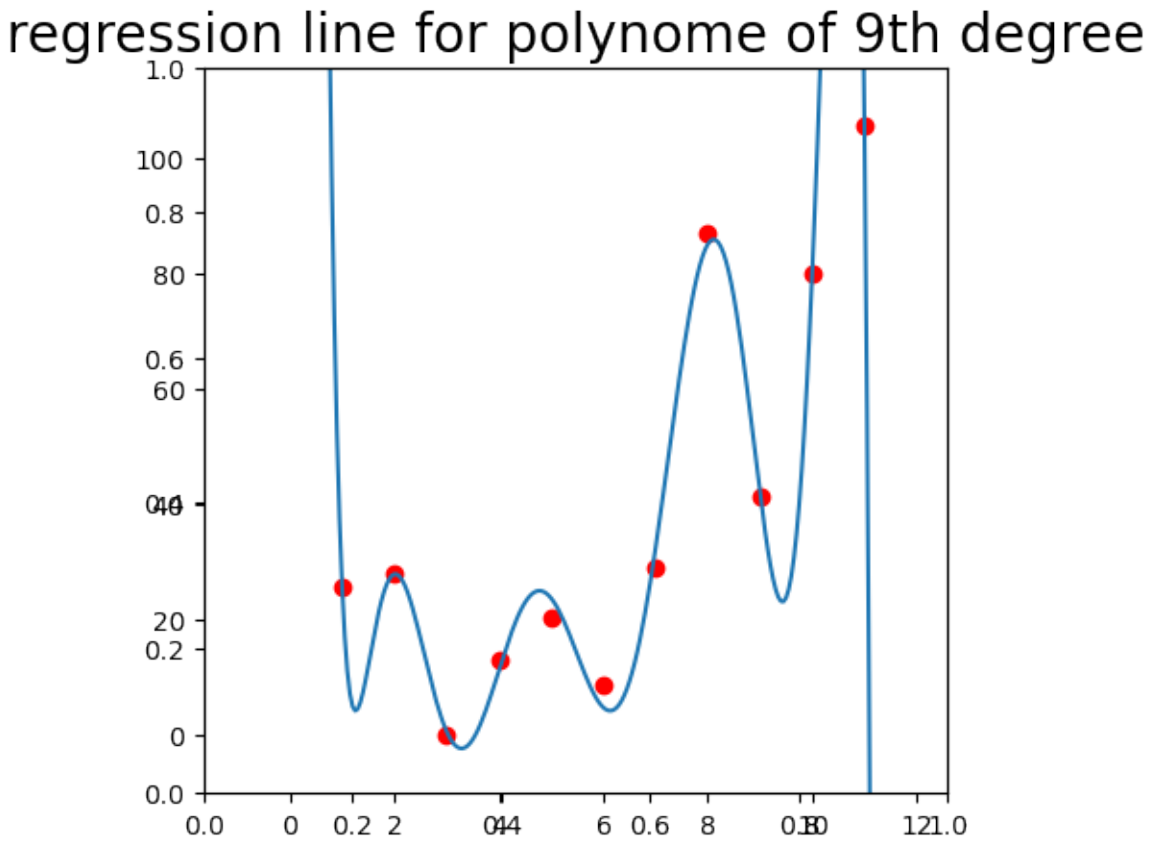
Nun wird diese Vorgehensweise für weitere Terme höherer Ordnung angewendet. Graphisch lässt sich zeigen, dass die Anpassung des Modells an die Daten immer besser wird, die Vorhersage für **neue Datenpunkte** aber sehr schlecht sein dürfte. Das Polynom hat an vielen Stellen Schlenker und absurde Kurven eingebaut. Dies ist ein erstes Beispiel für **"overfitting"**.Einen 'perfekten' fit erhält man, wenn man genausoviele Paramter (10 Steigunskoeffizienten + intercept) hat wie Daten-Messpunkte.

The important points to note here:

- the fit to our empirical y-values gets better
- at the same time, the regression line starts behaving strangly
- the predictions made by the regression line in between the empirical y-values are grossly wrong: this is an example of **overfitting**

```
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')
# underdetermined, ill-posed: infinitely many solutions
X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7, X**8, X**9]
x = np.arange(-1, 12, 0.05).reshape((-1, 1))
x = np.c_[x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9]
model.fit(X, y)
y_hat = np.dot(x , model.coef_.T)  + model.intercept_
```

```
(-10.0, 115.77315979942053)
```



5.2 perfect fit: as many variables as data samples

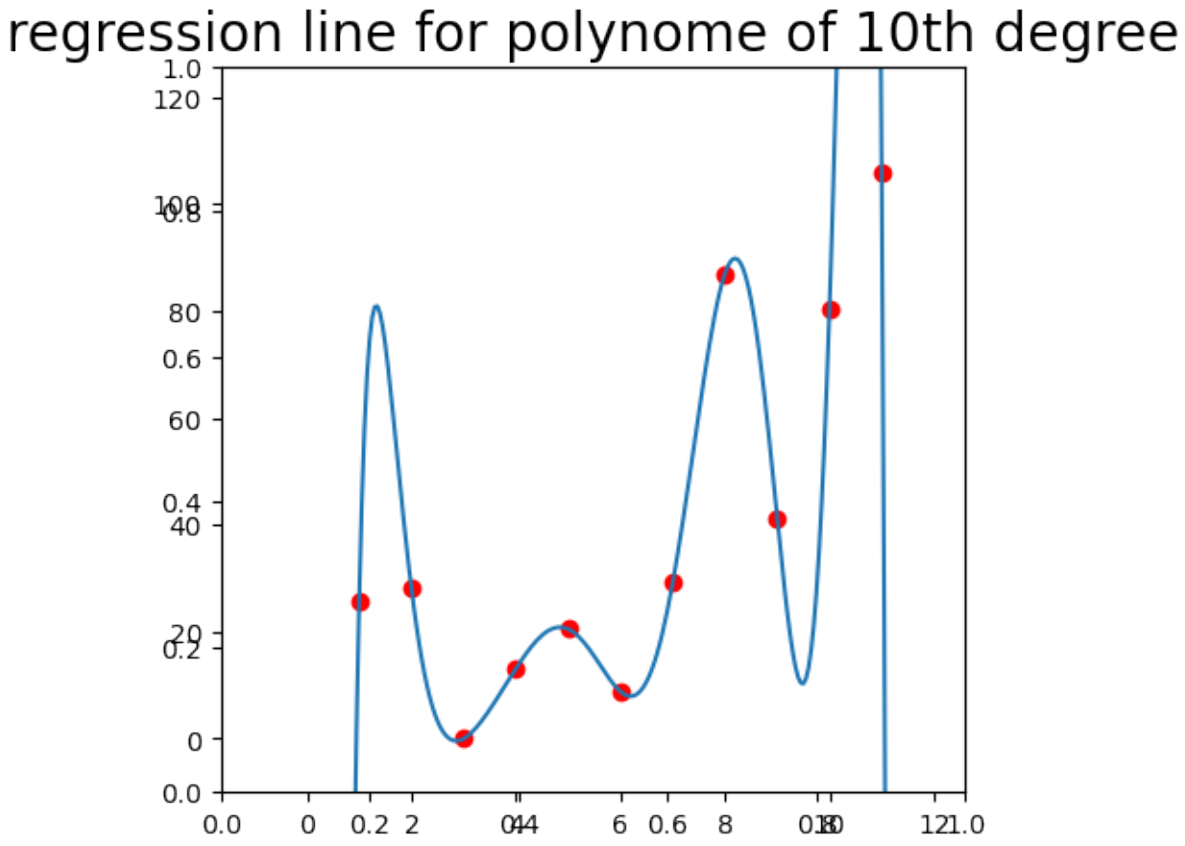
A perfect fit is possible as is demonstrated next. We have as many variables (terms derived from x) as observations (data points). So for each data point we have a variable to accommodate it. **Note**, that a perfect fit is achieved with 10 variables + intercept. The intercept is also a parameter and in this case the number of observations n equals the number of variables p, i.e. $p = n$.

```
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')
# underdetermined, ill-posed: infinitely many solutions
X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7, X**8, X**9, X**10]
x = np.arange(-1, 12, 0.05).reshape((-1, 1))
x = np.c_[x, x**2, x**3, x**4, x**5, x**6, x**7, x**8, x**9, x**10]
model.fit(X, y)
y_hat = np.dot(x , model.coef_.T)  + model.intercept_
```

```
print(f'the intercept and the coefficients are: {model.intercept_}, {model.coef_}')
```

```
the intercept and the coefficients are: [-3441.3761578], [[ 9.78847039e+03 -1.
↪13028575e+04  7.22272630e+03 -2.87529040e+03
   7.50863939e+02 -1.30675765e+02  1.49834150e+01 -1.08409478e+00
   4.47395935e-02 -8.00879370e-04]]
```

```
(-10.0, 125.77315979942053)
```



CHAPTER

SIX

WHAT HAPPENS IF WE HAVE MORE VARIABLES THAN DATA POINTS?

Gibt es mehr Parameter als Datenpunkte, existieren unendlich viele Lösungen und das Problem ist nicht mehr eindeutig lösbar. Früher gelang die Inversion des Kreuzproduktes der Variablen $\mathbf{X}'\mathbf{X}$ nicht. Mittlerweile gibt es Näherungsverfahren, die dennoch Ergebnisse liefern - wenn auch sehr Ungenaue.

Mittlerweile gibt es aber mathematische Näherungsverfahren die es ermöglichen auch singuläre Matrizen zu invertieren. `numpy` verwendet hierfür die sogenannte LU-decomposition.

One way to see in python that the solution is erroneous is to use the `scipy.linalg.solve` package and solve for the matix S that solves $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{S} = \mathbf{I}$. $\mathbf{I}$ is called the eye-matrix wih 1s in the diagonale and zeros otherwise:

$$\mathbf{I} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

Die entscheidende Zeile im nachfolgenden Code ist: `S = solve(inv(np.dot(X.T, X)), np.eye(13))`

Sie besagt: gib mir die Matrix $\mathbf{S}$, die multipliziert mit $(\mathbf{X}'\mathbf{X})^{-1}$ die Matrix $\mathbf{I}$ gibt. Für unseren Fall von mehr Variablen als Beobachtungspunkten werden wir gewarnt, dass das Ergebnis falsch sein könnte. Mit älteren Mathematik- oder Statistik-Programmen ist dies überhaupt nicht möglich.

```
warnings.filterwarnings("default")
from numpy.linalg import inv
from scipy.linalg import solve
model = LinearRegression()
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')

# underdetermined, ill-posed: infinitely many solutions
X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7, X**8, X**9, X**10, X**11, X**12,␣
 ↪X**13]

# this should give at least a warning, because matrix inversion as done above is not␣
 ↪possible
# any more, due to singular covariance matrix [X'X]
model.fit(X, y)
#y_hat = np.dot(x , model.coef_.T)  + model.intercept_
S = solve(inv(np.dot(X.T, X)), np.eye(13))
```

```
/home/martin/miniconda3/envs/book/lib/python3.7/site-packages/ipykernel_launcher.
 ↪py:15: LinAlgWarning: Ill-conditioned matrix (rcond=3.8573e-21): result may not be␣
 ↪accurate.
  from ipykernel import kernelapp as app
```

6.1 statistical package R

In der statistischen Programmiersprache R wird keine Warnung herausgegeben. Es werden einfach nur soviele Koeffizienten (intercept ist auch ein Koeffizient) berechnet, wie möglich ist. Alle weiteren Koeffizienten sind `NA`.

```
/home/martin/miniconda3/envs/book/lib/python3.7/site-packages/ipykernel/ipkernel.
↪py:287: DeprecationWarning: `should_run_async` will not call `transform_cell`␣
↪automatically in the future. Please pass the result to `transformed_cell` argument␣
↪and any exception that happen during thetransform in `preprocessing_exc_tuple` in␣
↪IPython 7.17 and above.
  and should_run_async(code)
```

```
>
> colnames(X)
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11" "V12"
[13] "V13" "y"
> lm("y ~ .",X)

Call:
lm(formula = "y ~ .", data = X)

Coefficients:
(Intercept)           V1           V2           V3           V4           V5
 -3.442e+03    9.790e+03   -1.130e+04    7.224e+03   -2.876e+03    7.510e+02
         V6           V7           V8           V9          V10          V11
 -1.307e+02    1.499e+01   -1.084e+00    4.474e-02   -8.010e-04           NA
        V12          V13
         NA           NA

```

CHAPTER

SEVEN

DEALING WITH OVERFITTING

Wie wir gesehen haben tendiert klassische Lineare Regression zu 'overfitting' sobald es wenige Datenpunkte gibt und mehrere Koeffizienten berechnet werden. Eine Lösung für dieses Problem ist, die Koeffizienten $b_1, b_2, b_3, ...$ kleiner zu machen. Dies kann erreicht werden, wenn der Fehler der Regression mit grösseren Koeffizienten auch grösser wird. Um nun das Minimum der Fehlerfunktion zu finden ist ein probates Mittel, die Koeffizienten kleiner zu machen und somit implizit 'overfitting' zu verhindern. Parameter können jetzt nur noch sehr gross werden, wenn dadurch gleichzeitig der Fehler stark reduziert werden kann.

Nachfolgend wird ein Strafterm ('penalty') für grosse Parameter eingeführt. Im Falle der Ridge-Regression gehen die Koeffizienten quadriert in die Fehlerfunktion mit ein. Der Gewichtungsfaktor λ bestimmt die Höhe des Strafterms und ist ein zusätzlicher Parameter für den – je nach Datensatz – ein optimaler Wert gefunden werden muss.

7.1 Ridge regression

Remember this formula:

$$\sum_i^n (y_i - \hat{y_i})^2 = \sum_i^n [y_i - (a + b \cdot x_i)]^2$$

To make the error term bigger, we could simply add $\lambda \cdot b^2$ to the error:

$$\sum_i^n (y_i - \hat{y_i})^2 + \lambda b^2 = \sum_i^n [y_i - (a + b \cdot x_i)]^2 + \lambda b^2$$

The parameter λ is for scaling the amount of shrinkage. Die beiden Ausdrücke

$$\sum_i^n [y_i - (a + b \cdot x_i)]^2 \tag{7.1}$$

$$\lambda b^2 \tag{7.2}$$

sind wie Antagonisten. Der Koeffizient b darf nur gross werden, wenn er es vermag (7.1) stark zu verkleinern, so dass der Zugewinn in (7.1) den Strafterm in (7.2) überwiegt.

For two variables we can write:

$$\sum_i^n (y_i - \hat{y_i})^2 + \lambda b_1^2 + \lambda b_2^2 = \sum_i^n [y_i - (a + b_1 \cdot x_{i1} + b_2 \cdot x_{i2})]^2 + \lambda b_1^2 + \lambda b_2^2$$

And in matrix notation for an arbitrary number of variables:

$$\min = (\mathbf{y} - \hat{\mathbf{y}})^2 + \lambda \mathbf{b}^2 = (\mathbf{y} - \mathbf{Xb})'(\mathbf{y} - \mathbf{Xb}) + \lambda \mathbf{b}'\mathbf{b}$$

Interessanterweise gibt es für diesen Fall ebenfalls eine exakte analytische Lösung. Allerdings haben wir den intercept Koeffizienten a mit in $\mathbf{b}$ aufgenommen und die zusätzliche Spalte mit lauter Einsern in $\mathbf{X}$ hinzugefügt. Wenn wir nun

$\lambda \mathbf{b}'\mathbf{b}$ berechnen, den quadrierten Strafterm für den Parametervektor, dann würden wir auch a bestrafen. Die Rolle von a ist aber, die Höhenlage der Regressionsfunktion zu definieren (die Stelle an der die Funktion die y-Achse schneidet). Der intercept a kann allerdings aus der Gleichung genommen werden, wenn die Variablen vorher standardisiert werden (Mittelwert $\bar{x} = 0$ und $\bar{y} = 0$). Jetzt verschwindet a von ganz allein, wenn wir die standardisierten Mittelwerte in die Gleichung für a einfügen:

$$a = \bar{y} - b\bar{x} = 0 - b \cdot 0 = 0$$

Nun muss a nicht mehr berücksichtigt werden und die Lösung für $\mathbf{b}$ ergibt sich zu:

$$\hat{\mathbf{b}} = (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}'\mathbf{y}$$

Nach Hastie et al., wurde dieses Verfahren ursprünglich verwendet um 'rank deficiency' Probleme zu beheben. Wenn Die Spalten oder Zeilen einer Matrix nicht lineare unabhängig sind, so hat die Matrix nicht vollen Rang. Beispielsweise kann sich eine Spalte durch Addition anderer Spalten ergeben. In diesem Fall funktionierte die Matrix Inversion nicht zufriedenstellend. Als Lösung hat man gefunden, dass es ausreichend ist, einen kleinen positiven Betrag zu den Diagonal-Elementen der Matrix zu addieren. Dies wird nachfolgend in einem numerischen Beispiel gezeigt:

- `np.c_` fügt die einzelenne Variablen zu einer Matrix zusammen
- `np.dot(X.T, X)` ist das bekannte Kreuzprodukt der transponierten Matrix $\mathbf{X}'$ und $\mathbf{X}$
- `np.linalg.matrix_rank` gibt uns den Rang der Matrix
- `np.eye(7) * 2` erstellt eine Diagonal-Matrix mit 2 in der Diagonalen und 0 überall sonst

```
X_6 = np.c_[X, X**2, X**3, X**4, X**5, X**6]
print(f'With 6 variables (polynom of 6th degree), the rank of the quare matrix\n is '\
      + f'{np.linalg.matrix_rank(np.dot(X_6.T, X_6))}')

X_7 = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7]
print(f'With 7 variables (polynom of 7th degree), the rank of the quare matrix\n is '\
      + f'{np.linalg.matrix_rank(np.dot(X_7.T, X_7))}')

print(f'By adding a small amount to the diagonal of the matrix, it is of full rank\n␣
 ↪again: '\
     + f'{np.linalg.matrix_rank(np.dot(X_7.T, X_7) + np.eye(7) * 2)}')
## you can see how small this amount is, by having a glimpse on the diagonal elements:
print('\nto see how small the added amount in reality is, we display the diagonal␣
 ↪elements:')
np.diag(np.dot(X_7.T, X_7))
```

```
With 6 variables (polynom of 6th degree), the rank of the quare matrix
 is 6
With 7 variables (polynom of 7th degree), the rank of the quare matrix
 is 6
By adding a small amount to the diagonal of the matrix, it is of full rank
 again: 7

to see how small the added amount in reality is, we display the diagonal elements:
```

```
array([           506,          39974,        3749966,      382090214,
        40851766526,   4505856912854, 507787636536686])
```

7.1.1 example of ridge regression

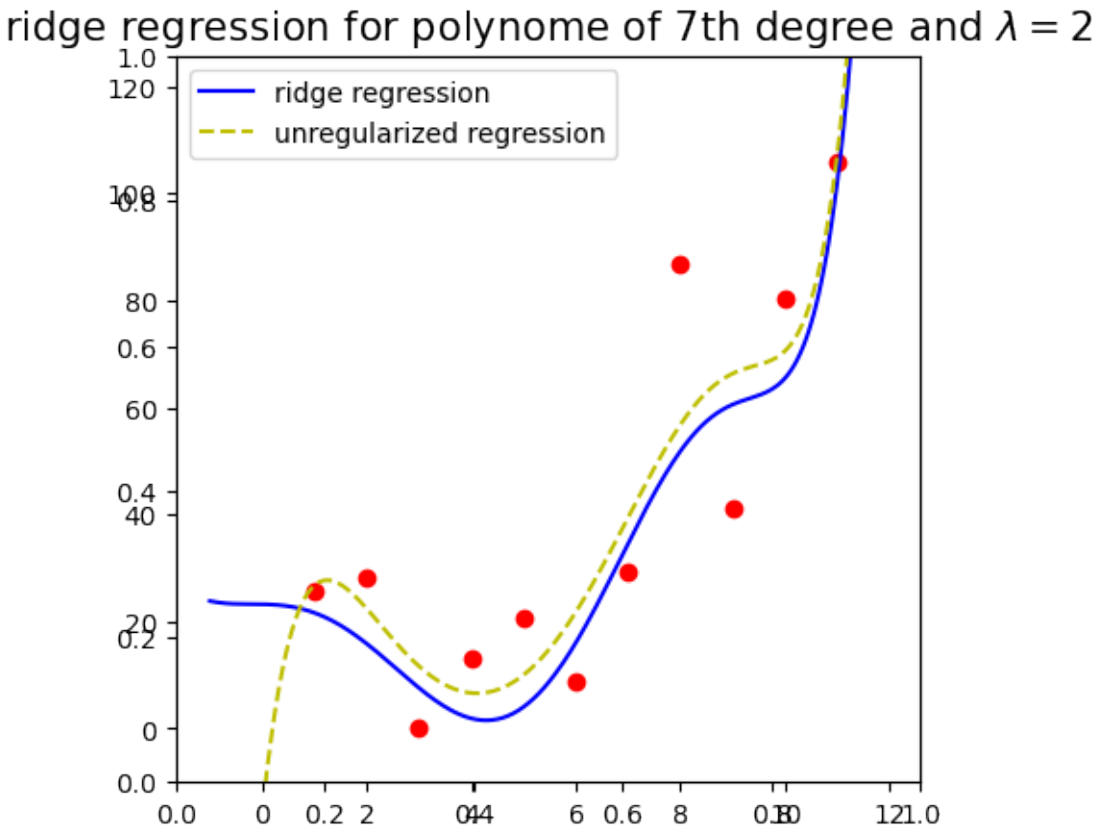
Next, we will apply ridge regression as implemented in the python `sklearn` library and compare the results to the linear algebra solution. Note, that we have to center the variables.

- we can center **X** and **y** and display the result in the centered coordinate system
- or we can center **X** and add the mean of **y** to the predicted values to display the result in the original coordinate system. This approaches allows for an easy comparison to the overfitted result

Die Zeile `Xc = X - np.mean(X, axis=0)` standardisiert die Variablen auf den Mittelwert von 0

```
from sklearn.linear_model import Ridge
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')

X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7]
# here is the necessary standardization:
Xc = X - np.mean(X, axis=0)

# for plotting purpose
x = np.arange(-1, 12, 0.05).reshape((-1, 1))
x = np.c_[x, x**2, x**3, x**4, x**5, x**6, x**7]
xc = x -np.mean(x, axis = 0)

# the result as obtained from the sklearn library
model = Ridge(alpha=2, fit_intercept=False)
model.fit(Xc, y)
print(f'the parameters from the sklearn library:\n'\
     + f'{model.coef_}')

# the analytical result as discussed above
inverse = np.linalg.inv(np.dot(np.transpose(Xc), Xc) + np.eye(Xc.shape[1]) * 2)
Xy = np.dot(np.transpose(Xc),y)
params = np.dot(inverse, Xy)
print(f'the parameters as obtained from the analytical solution:\n'
     + f'{np.transpose(params)}')
params_ridge = params
```

```
the parameters from the sklearn library:
[[-1.96523108e-01 -6.47914003e-01 -9.37247119e-01  1.55320112e-01
   3.20681202e-02 -6.80277139e-03  3.08899915e-04]]
the parameters as obtained from the analytical solution:
[[-1.96523119e-01 -6.47914004e-01 -9.37247118e-01  1.55320112e-01
   3.20681203e-02 -6.80277139e-03  3.08899915e-04]]
```



7.2 Lasso

Alternativ zu einem quadratischen Strafterm b^2 könnte man auch den absoluten Wert nehmen $|b|$. In diesem Fall erhält man die sog.~Lasso Regression; $\lambda \cdot |b|$ wird zum Vorhersage-Fehler addiert:

$$\sum_i^n (y_i - \hat{y_i})^2 + \lambda|b| = \sum_i^n [y_i - (a + b \cdot x_i)]^2 + \lambda|b|$$

Für zwei Variablen würde man folglich schreiben:

$$\sum_i^n (y_i - \hat{y_i})^2 + \lambda|b_1| + \lambda|b_2| = \sum_i^n [y_i - (a + b_1 \cdot x_{i1} + b_2 \cdot x_{i2})]^2 + \lambda|b_1| + \lambda|b_2|$$

Leider gibt es im Gegesatz zur Ridge Regression keine eindeutige analytische Lösung um die Koeffizienten der Lasso Regression zu erhalten. Hier kommen iterative Verfahren zum Einsatz, wie wir sie in Session 2 kennen lernen werden.

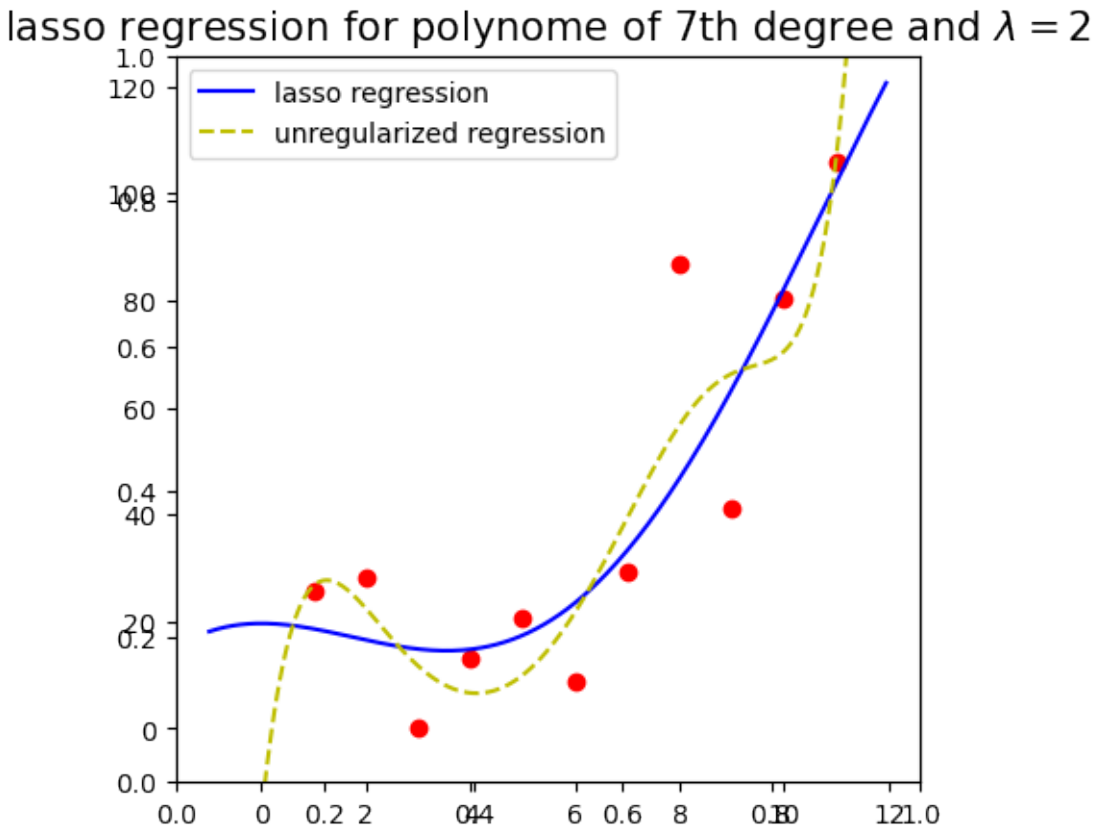
7.2.1 Vergleich der Koeffizienten der Lasso Regression mit denen der Ridge Regression

Next, we will apply lasso regression as implemented in the python sklearn library and compare the results to the unconstraint regression results. As before, we have to center the variables (-> see discussion above)

```
import numpy as np
from sklearn.linear_model import Lasso
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')

X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7]
Xc = X - np.mean(X, axis=0)

# for plotting purpose
x = np.arange(-1, 12, 0.05).reshape((-1, 1))
x = np.c_[x, x**2, x**3, x**4, x**5, x**6, x**7]
xc = x -np.mean(x, axis = 0)

# the result as obtained from the sklearn library
model = Lasso(alpha=2, fit_intercept=False)
model.fit(Xc, y)
params_lasso = model.coef_

# comparison of parameters ridge vs. lasso:
print(f'the parameters of the ridge regression:\n'\
     + f'{np.transpose(params_ridge)}')

print(f'the parameters of the lasso regression:\n'\
     + f'{params_lasso}')
```

```
the parameters of the ridge regression:
[[-1.96523119e-01 -6.47914004e-01 -9.37247118e-01  1.55320112e-01
   3.20681203e-02 -6.80277139e-03  3.08899915e-04]]
the parameters of the lasso regression:
[-0.00000000e+00 -1.27169261e+00  2.49755651e-01  7.47152651e-04
 -5.77539403e-04 -2.73002774e-05  1.76588437e-06]
```

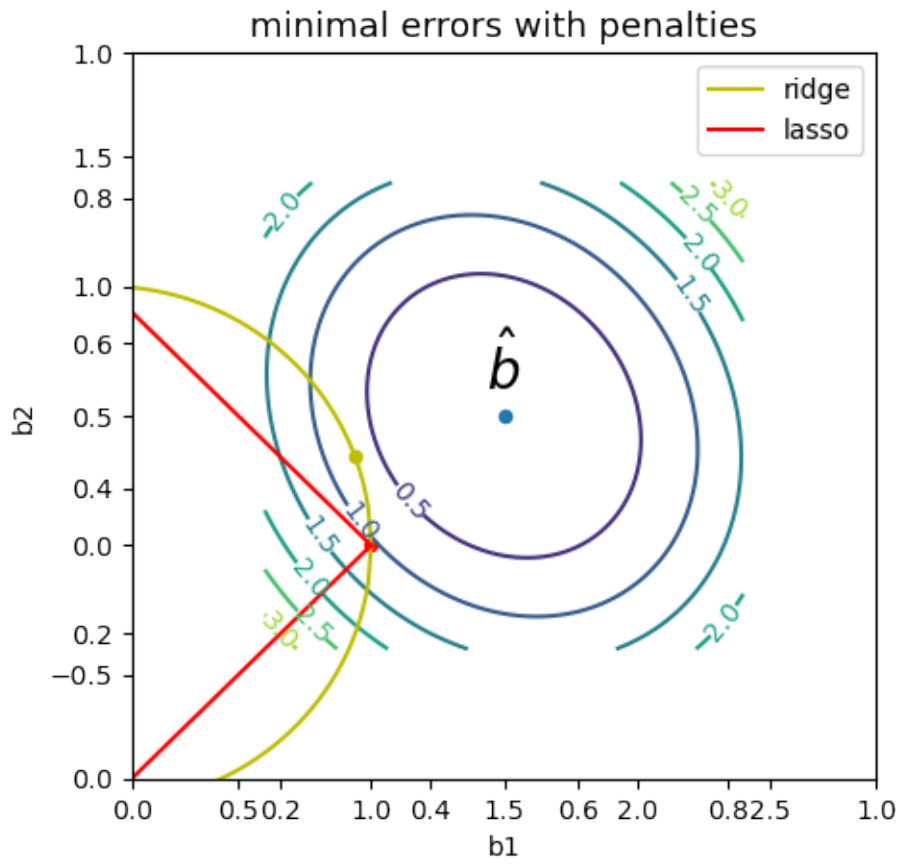
Ridge Regression tendiert dazu alle Koeffizienten im gleichen Mass zu verkleinern. Lasso führt oft zu Lösungen, bei denen einige Koeffizienten ganz zu 0 konvergiert sind. Wenn man die Ergebnisse im obigen Beispiel betrachtet, fällt einem auf dass für Lasso eigentlich nur zwei Koeffizienten verschieden von 0 sind (for X^2 and X^3). Die Werte alle anderen Koeffizienten sind kleiner als $0.000747 = 7.47\text{e} - 04$.



7.3 the difference between ridge and lasso

In der folgenden graphischen Darstellung haben die **wahren Koeffizienten** die Werte $b_1 = 1.5, \quad b_2 = 0.5$. Für ein grid aus beliebigen Werten für b_1 und b_2 wird der **mean squared error** (MSE) berechnet und der Fehler als Kontur graphisch dargestellt. Wie man sieht, wird der Fehler umso geringer, je näher die Koeffizienten im grid an den wahren Koeffizienten liegen. Als nächstes werden alle Koeffizienten-Kombinationen aus b_1 und b_2 eingetragen, deren Strafterm ($b_1^2 + b_2^2$ im Falle von Ridge und $b_1 + b_2$ im Falle von Lasso) den Wert von 1.0 nicht übersteigt. Die Lösung, die den **wahren Koeffizienten** am nähesten ist, wird jeweils durch einen Punkt eingezeichnet.

Hierbei sieht man, dass sich die besten Lösungen von Ridge auf einem Halbkreis bewegen, die von Lasso auf einem Dreieck. An der Stelle, an der die Lasso-Lösung der eigentlichen Lösung (b=1.5, b2=0.5) am Nähesten ist, ist ein Parameter (b_2) fast 0. Das zeigt die Tendenz von Lasso, einige Parameter gegen 0 zu schrumpfen. Dieses Verhalten kann man sich zum Beispiel bei Variablen-Selektion zu Nutzen machen.




```
optimal coefficients of the ridge solution: 0.9393939393939394 and 0.34283965148438655
optimal coefficients of the lasso solution: 1.0 and 0.0
```

CHAPTER

EIGHT

ELASTICNET

Aus der Physik kommend werden die Strafterme von Ridge und Lasso als L_2 und L_1 bezeichnet. Eigentlich ist die L_2-Norm die Quadratwurzel der Summe der quadrierten Elemente eines Vectors und die L_1-Norm nur die Summe der Vektorelemente. ElasticNet ist ein lineares Regressions-Verfahren, in welches sowohl die regularization-terms von Lasso (L_1), als auch von Ridge (L_2) eingehen. Hier gibt es nicht nur einen λ-Paramter, der das Ausmass von regularization bestimmt, sondern einen zusätzlichen Parameter α, der das Verhältnis von L_1 und L_2 regularization angibt.

Weil Ridge Regression und Lasso die Koeffizienten sehr unterschiedlich regulieren, ist als Kompromiss die Kombination aus beiden Methoden sehr beliebt geworden.

$$\lambda \sum_j (\alpha b_j^2 + (1-\alpha)|b_j|)$$

Die Interpretation der beiden paramter λ und α ist wie folgt:

- λ bestimmt das generelle Mass an regularisation
- α gibt das Verhältnis an, mit dem diese beiden Strafterme indie regularisation einfliessen sollen

Im Übungs-Notebook zu den Boston house-prices werden wir ElasticNet verwenden.

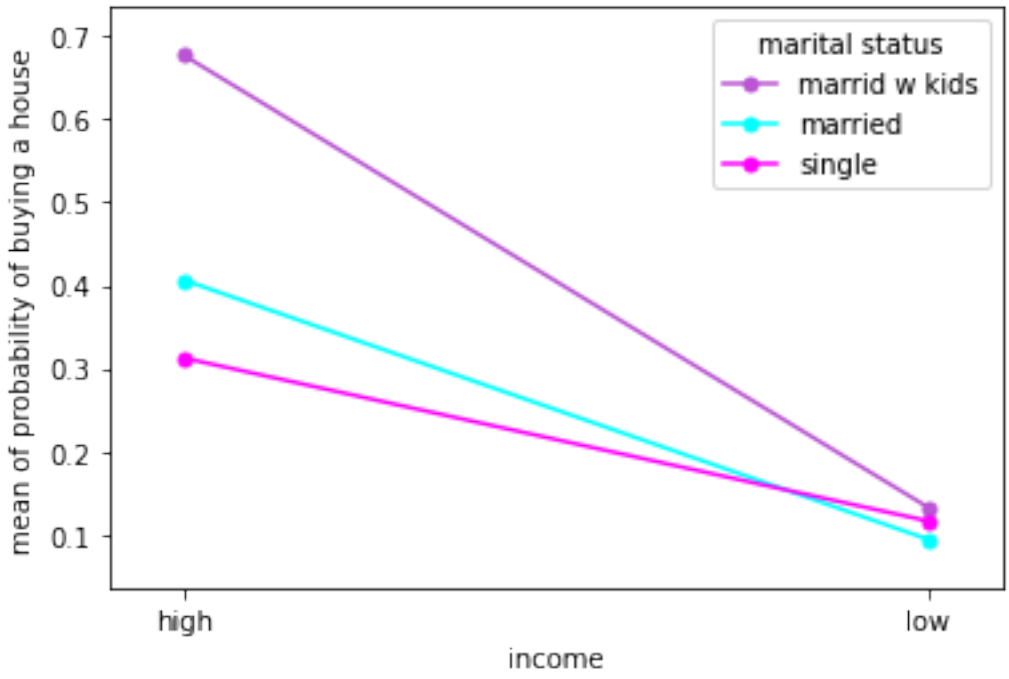
CHAPTER

NINE

INTERACTION

Interaktionen sind ein weiteres wichtiges Konzept in der linearen Regression. Hier ist der Effekt einer Variablen auf die abhängige Variable y abhängig von dem Wert einer anderen Variable.

In unterem Beispiel versuchen wir die Wahrscheinlichkeit zu modellieren, dass eine Person ein Haus kauft. Natürlich ist das monatliche Einkommen eine wichtige Variable und desto höher dieses, desto wahrscheinlicher auch, dass besagte Person ein Haus kauft. Eine andere wichtige Variable ist der Zivilstand. Verheiratet Personen mit Kindern im Haushalt tendieren stark zu Hauskauf, besonders wenn das monatliche Einkommen hoch ist. Auf der anderen Seite werden Singles, auch wenn sie ein hohes Einkommen haben, eher nicht zum Hauskauf tendieren. Wir sehen also, die Variable "monatliches Einkommen" **interagiert** mit der Variable "Zivilstand":



Das obige Beispiel beinhaltete kategorielle Variablen. Beispiele wie diese trifft man oft im Bereich der Varianzanalysen (ANOVA) an. Interaktions-Effekte bestehen aber auch für kontinuierliche Variablen. In diesem Fall ist es aber etwas komplizierter die Effekte zu visualisieren. Wir werden jetzt unseren eigenen Datensatz so erzeugen, dass er einen deutlichen Interaktions-Effekt aufweist. Damit der Effekt zwischen 2 kontinuierlichen Variablen überhaut in 2D dargestellt werden kann, musss eine der beiden Variablen wieder diskretisiert werden, d.h. wir müssen für sie wieder Kategorien bilden. Im nächsten Rechenbeispiel versuchen wir dann, die Parameter, die zur Generierung der Daten gedient haben mit einer Linearen-Regressions-Analyse wieder zu finden. Die Daten wurden nach folgendem Modell generiert:

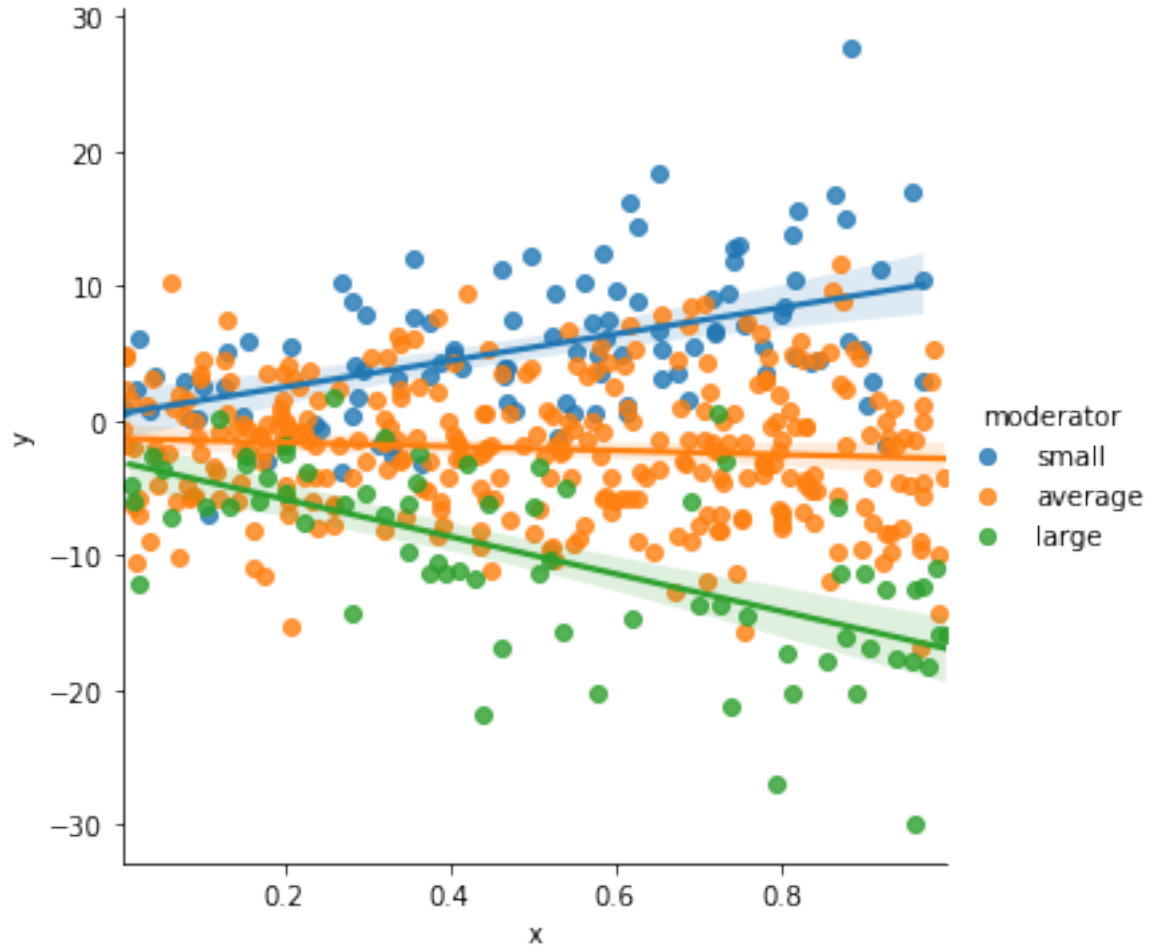
$$y = 2 \cdot x + -2 \cdot m + -7 \cdot (x \cdot m) + \text{np.random.normal(loc = 0, scale = 4, size = n)}$$

`np.random.normal(loc=0, scale=4, size=n)` ist der Random-Error-Term, den wir hinzufügen, damit die Daten nicht alle auf einer Lienie liegen. `loc=0` besagt, dass der Mittelwert unseres zufälligen Fehlers 0 ist, `scale=4`, dass die Varianz der Werte 4 ist und `size=n` gibt die Anzahl der zu generierenden zufälligen Werte an

Folgliche haben wir also die Koeffizienten:

- $b_x = 2$
- $b_m = -2$
- $b_{x \cdot m} = -7$

```
import seaborn as sns
n = 500
x = np.random.uniform(size=n)
m = np.random.normal(loc = 0.5, scale = 1, size = n)

#  lin effects + interaction + random error
y = 2*x + -2*m + -7*(x*m) + np.random.normal(loc = 0, scale = 4, size = n)

newM = pd.cut(m, bins=3, labels = ['small', 'average', 'large'])


toy = pd.DataFrame({'x' : x, 'y' : y, 'moderator' : newM})
sns.lmplot(x="x", y="y", hue="moderator", data=toy);
```



Interaktions-Terme können gebildet werden, indem man zwei Variablen elemente-weise miteinander multipliziert. Durch die Hinzuname weiterer Terme sollte die Modell-Anpassung eigentlich besser werden - besonders wenn ein starker Interaktionsterm in den Daten vorliegt, so wie wir ihn eingebaut haben. Vergleichen wir die Koeffizienten, so wie sie im Linearen-Modell gefunden werden mit denen, die zur Erzeugung unseres Datensatzes gedient haben. Gar nicht schlecht, oder? Die zufälligen Fehler mit der grossen Varianz sorgen natürlich dafür, dass sie dennoch von den 'generating parameters' verschieden sind.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
X = np.c_[x, m]
model.fit(X, y)
y_hat = model.intercept_  + np.dot(X, model.coef_)
print(f'without considering the interaction, the mse is: {np.mean((y-y_hat)**2)}')

X = np.c_[x, m, x * m]
model.fit(X, y)
y_hat = model.intercept_  + np.dot(X, model.coef_)
print(f'considering the interaction, the mse drops to: {np.mean((y-y_hat)**2)}')
print(f'\nthe coefficients are given by {model.coef_}; compare these values\n to the␣
↪values '\
    + f'we used for generating the data')
```

```
without considering the interaction, the mse is: 21.027778912546122
considering the interaction, the mse drops to: 15.46963918621246

the coefficients are given by [ 2.30055754 -1.20686064 -7.89087321]; compare these␣
↪values
 to the values we used for generating the data
```

9.1 some considerations

Die Überlegung hier veranschaulicht, dass es schon bei moderater Variablen-Anzahl sehr viele mögliche Interaktions-Terme gibt. Für die normale Lineare Regression würde die grosse Anzahl dieser Terme zum Verhängnis werden, weil dann wieder der Fall eintreten könnte indem wir die Daten overfitten oder gar mehr Variablen als Beobachtunge zur Verfügung stehen. Auch in diesem Fall kann auf die vorgestellten^ Regularisierungs-Verfahren (ElasticNet, Ridge und Lasso) zurückgegriffen werden:

Nehmen wir an, wir haben ein data-set mit 70 verschiedenen Variablen. Weil wir nichts über die Beziehungen der Variablen zur abhängigen Variable y noch über die Beziehungen der Variablen untereinander wissen, sind wir geneigt eine Menge zusätzlicher 'features' für unser Modell zu erzeugen:

- wir können 70 quadratische Terme hinzufügen (x_j^2)
- wir können 70 kubische Terme aufnehmen (x_j^3)
- wir können auch $\binom{70}{2} = 2415$ Interaktionen erster Ordnung zwischen den 70 Variablen annehmen
- anstatt dessen könnte wir auch die Interaktions-Terme der 210 (70 Variablen + 70 quadratische Terme + 70 kubische Terme) Variablen mit aufnhemne: $\binom{210}{2} = 21945$
- neben quadratisch und kubischen Termen gibt es auch viele andere linearisierende Transformation, die unter Umständen zu besseren ergebnissen führen wie beispielsweise die log-Transformation. Im praktischen Beipiel des Bosten house-prices data-Sets werden wir die `box-cox-Transformation` kennen lernen.

Wie wir gesehen haben, kann die Anzahl möglicher Variablen sehr schnell wachsen, wenn man alle Effekte berücksichtigt, die ausschlaggebend sein könnten. Manchmal existieren sogar Interaktionseffekte zweiter Ordnund, d.h. drei Variablen

sind dann daran beteiligt. Würden wir alle möglichen Variablen berücksichtigen, die sich derart bilden lassen, dann würde dies auch bei grossen Daten-Sets zu ausgeprägten ‘overfitting’ führen. **Aus diesem Grund wurden die regularization techniques wie das ElasticNet und seine Komponenten, die Ridge Regression und die Lasso Regression eingeführt**.

CHAPTER

TEN

WIE ZUVERSICHTLICH SIND WIR HINSICHTLICH UNSERER MODELL-VORHERSAGEN

Selten werden wir mit unserem Modell genau die Koeffizienten schätzen können, die in der gesamten Population (alle Daten, die wir erheben könnten) anzutreffen sind. Viel öfter ist unsere Stichprobe nicht repräsentativ für die gesamte Population oder sie ist schlicht zu klein und zufällige, normalverteilte Fehler in unseren Daten beeinflussen die Schätzung der Koeffizienten. Dies umsomehr, desto mehr Variablen wir in user Modell aufnehmen. Wie können wir nun die Güte unserer Schätzung beurteilen? Hier sind mindestens zwei verschieden Fragen denkbar:

- Wie sicher sind wir mit Hinblick auf die geschätzen Koeffizienten **b**?. Diese Frage ist besonders für Wissenschaftler wichtig, da die Antwort dafür ausschlaggebend ist, ob eine Hypothese beibehalten oder verworfen werden muss.
- Wie sicher sind wir uns bezüglich einzelner Vorhersagen. Dies spielt die grösste Rolle im Machine Learning Umfeld, da wir das trainierte Modell gerne in unsere Business-Abläufe integrieren würden.

Diese beiden Fragestellungen lassen sich mit Hinblick auf die Regression auch wie folgt formulieren:

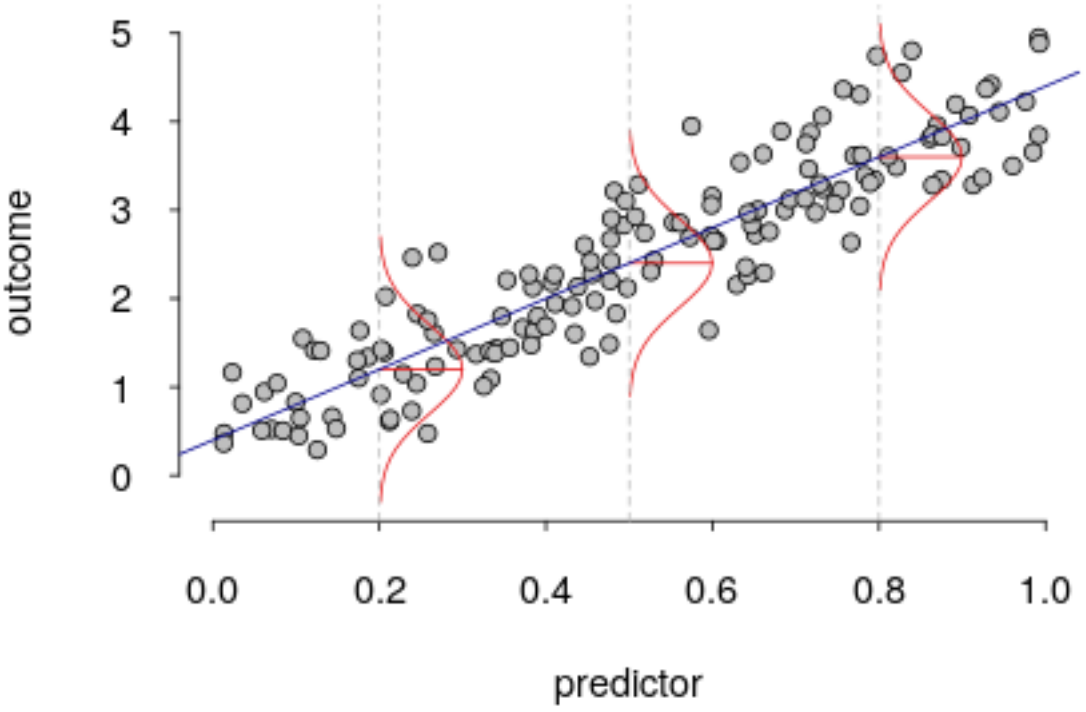
- Wie sehr ist die 'mean response', unsere Regressions-Funktion von der Stichprobe abhängig. Variiert Erstere sehr stark und umfasst unter Umständen sogar den Wert 0, dann können diese Effekte (Koeffizienten) nicht interpretiert werden.
- Wie sehr können Beobachtungen y für eine gegebene Kombination von Variablen-Werten in $\mathbf{X}$ variieren? Ist diese Variation sehr gross, so werden wir auch grosse Fehler in unseren Business-Process einbauen

10.1 Recap of assumptions underlying regression

Dies sind Linearität (der Zusammenhang einer Variablen und der abhängigen Variablen ist linear, d.h. der selbe Steigungsparamter gilt für alle Bereiche der Variablen), Homoskedastizität (die Fehler der Regression – die Residuen – sind in allen Bereichen von X normal verteilt mit gleicher Varianz) und Normalität der Residuen bei gegebenem Wert von X. Diese Voraussetzungen sind in vielen Fällen nicht erfüllt und auch bekannterweise verletzt.

- **Linearity**: Die Regression-Funktion ist eine gute Annäherung für die Beziehung zwischen $\mathbf{X}$ and $\mathbf{y}$, d.h. ist ein quadratischer Trend in den Daten und wir haben keine quadratischen Effekte in das Modell aufgenommen, so sind die Annahmen nicht erfüllt. Die Linearität besagt nämlich, dass für den Zusammenhang einer Variablen x und der abhängigen Variablen y der selbe Steigungs-Koeffizient b_x für all Bereich für x gelten muss. Ansonsten hat das Modell einen **bias**, es schätzt einen Koeffizienten systematisch falsch.
- **Homoscedasticity**: Die Varianz unseres Vorhersagefehlers (Residuen) ist für alle Bereiche einer Variablen x identisch.
- **Normality**: Die Werte der abhängigen Variablen $\mathbf{y}$ sind für einen gegeben Wert von $\mathbf{x}$ normal verteilt: $\mathbf{y}|\mathbf{x} \sim N(\mu, \sigma)$

In der nächsten Graphik werden die Voraussetzungen der linearen Regression veranschaulicht: Image taken from here



Now, with respect to our confidence need:

1. **Prediction interval**: The interval around our prediction, 95% (97.5%) of all observed values are supposed to fall in; This interval is symmetrical around the regression line. This fact follows from the assumptions discussed above. The standard error of prediction (or forecast) is given by: $\hat{\sigma}_e = \sqrt{\frac{1}{N-(p+1)} \sum_i^N e_i^2}, with p being the number of parameters (the term +1 is for the intercept); e_i are the residuals, i.e., the diffe
$y_i \pm t_{1-\alpha/2, N-p} \cdot \hat{\sigma}_e$. Here, t{1-\alpha/2, N-p} is the value of the student $-$ t $-$ distribution for a confidence level of 1-\alpha/2 and N-p$ degrees of freedom.

1. **Confidence interval**: In a similar manner (a bit more involved) we could derive the confidence interval for the predicted mean $\hat{y}_i$. Remember, that data is supposed to be normally distributed. The regression line we fit, is an estimate of the mean for a given configuration $\mathbf{x}_i$. Of course, we do not fit the empirical values exactly; some may be lying above the regression line, some beneath. This confidence interval gives an upper and a lower bound for the mean estimate, i.e. the regression line. This confidence interval is not equidistant from the regression line for all values of **x**. In the regions where data is sparse, the regression line can not be estimated with high confidence. In contrast, near the mean of **x** the estimate is supposed to be more accurate (normaly distributed **x** assumed).

1. **CI for regression coefficients**: Again, the derivation of the formulae for this CI is more involved than this for the prediction interval. This interval gives the upper and lower boundary for the coefficients **b**. These coefficients indicate how important the respectiv variable is in the regression equation. The interpretation of these coefficients is linked to `real` science, where the epistemological caveat is the matter of interest. For example: "is closing schools and universities related to lower base reproduction numbers (R_0)". This is typically not the kind of questions a data scientist is trying to answer ;-)

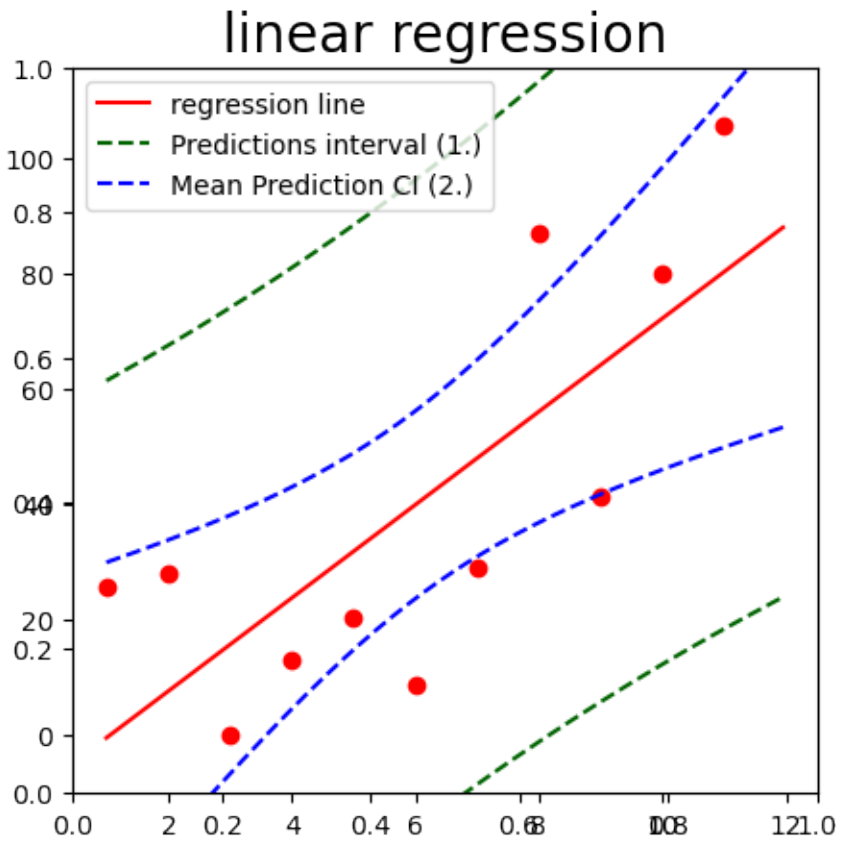
In the following code examples, first, we display the classical summary statistics. In the middle of the printed output, you can find the confidence intervals for the regression coefficients 'const' (intercept) and x_1, the b_1 coefficient. The plots illustrate te he points 1 and 2.

If someone has a strong interest in these more statistical models, I can recommend this source.

```
import statsmodels.api as sm

# data example
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')

# the x (small x) is just for plotting purpose
x = np.arange(1, 12, 0.05).reshape((-1, 1))
x_intercept = np.c_[np.ones(x.shape[0]), x]


X_intercept = np.c_[np.ones(X.shape[0]), X]

ols_result_lin = sm.OLS(y, X_intercept).fit()
y_hat_lin = ols_result_lin.get_prediction(x_intercept)



dt_lin = y_hat_lin.summary_frame()
mean_lin = dt_lin['mean']
meanCIs_lin = dt_lin[['mean_ci_lower', 'mean_ci_upper']]
obsCIs_lin = dt_lin[['obs_ci_lower', 'obs_ci_upper']]
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.584
Model:                            OLS   Adj. R-squared:                  0.538
Method:                 Least Squares   F-statistic:                     12.64
Date:                Wed, 14 Jul 2021   Prob (F-statistic):            0.00616
Time:                        22:43:16   Log-Likelihood:                -49.385
No. Observations:                  11   AIC:                             102.8
Df Residuals:                       9   BIC:                             103.6
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -8.4903     15.410     -0.551      0.595     -43.351      26.370
x1             8.0791      2.272      3.556      0.006       2.939      13.219
==============================================================================
Omnibus:                        4.018   Durbin-Watson:                   1.670
Prob(Omnibus):                  0.134   Jarque-Bera (JB):                1.165
Skew:                           0.156   Prob(JB):                        0.559
Kurtosis:                       1.437   Cond. No.                         14.8
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

```
(-10.0, 115.77315979942053)
```



The same plot is derived for an equation including a quadratic term:

```
X_intercept_quad = np.c_[X_intercept, X**2]

# for plotting:
x = np.arange(1, 12, 0.05).reshape((-1, 1))
x_intercept_quad = np.c_[np.ones(x.shape[0]), x, x**2]

ols_result_quad = sm.OLS(y, X_intercept_quad).fit()


y_hat_quad = ols_result_quad.get_prediction(x_intercept_quad)
dt_quad = y_hat_quad.summary_frame()
mean_quad = dt_quad['mean']
meanCIs_quad = dt_quad[['mean_ci_lower', 'mean_ci_upper']]
obsCIs_quad = dt_quad[['obs_ci_lower', 'obs_ci_upper']]
```

```
print(ols_result_quad.summary())
```

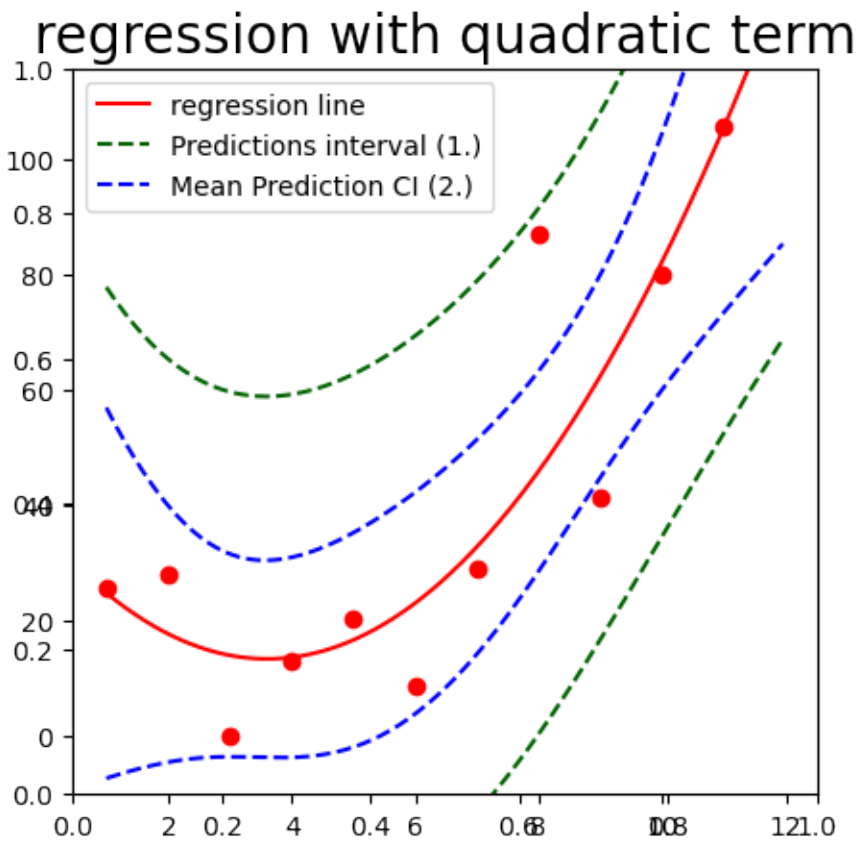
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.783
Model:                            OLS   Adj. R-squared:                  0.728
Method:                 Least Squares   F-statistic:                     14.39
Date:                Wed, 14 Jul 2021   Prob (F-statistic):           0.00224
Time:                        22:43:17   Log-Likelihood:                -45.820
No. Observations:                  11   AIC:                             97.64
```

(continues on next page)

(continued from previous page)

```
Df Residuals:                        8   BIC:                             98.83
Df Model:                            2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         35.3379     20.074      1.760      0.116     -10.952      81.628
x1           -12.1493      7.688     -1.580      0.153     -29.879       5.580
x2             1.6857      0.624      2.701      0.027       0.247       3.125
==============================================================================
Omnibus:                        9.915   Durbin-Watson:                   2.828
Prob(Omnibus):                  0.007   Jarque-Bera (JB):                4.642
Skew:                           1.313   Prob(JB):                       0.0982
Kurtosis:                       4.798   Cond. No.                         234.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
 ↪specified.
```

```
(-10.0, 115.77315979942053)
```

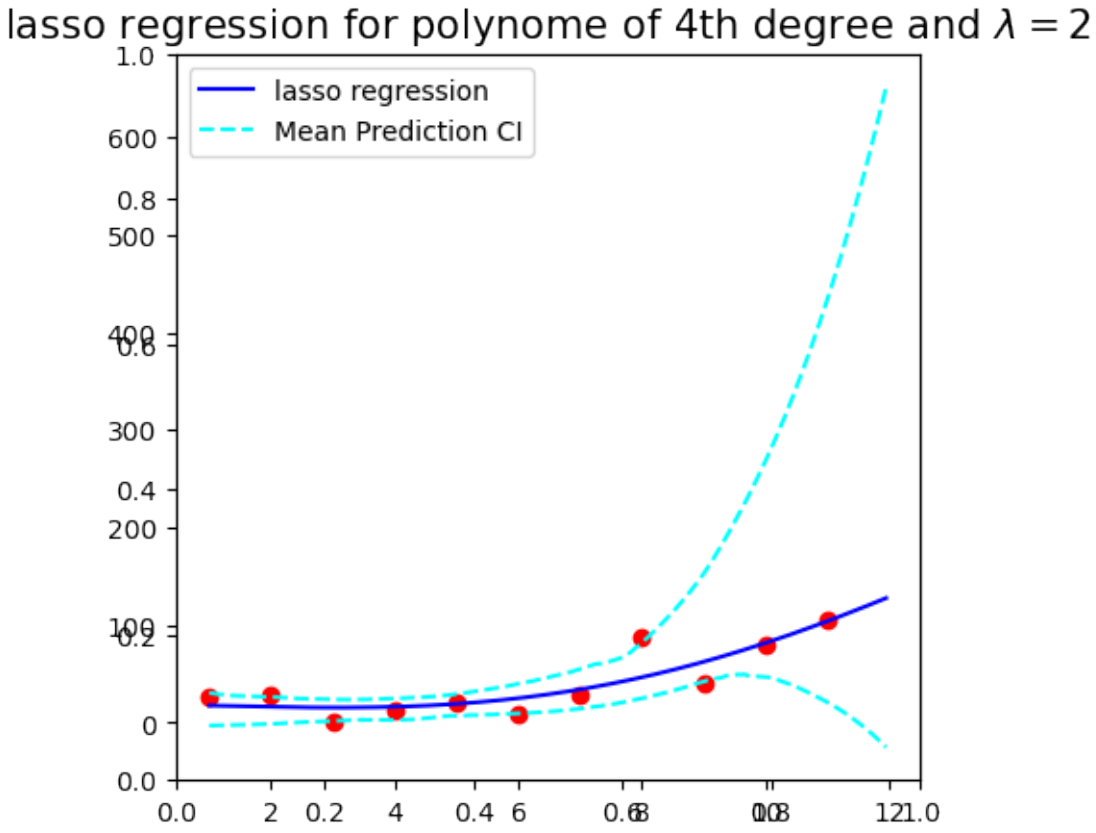


10.2 Bootstrap

With real, messy data it is rather seldom to meet all the assumptions underlying the theory of confidence intervals. A robust alternative, without any assumptions is the bootstrap. We view our data sample as the population and draw samples from it, with replacement. We fit the model to each of these samples and gather the statistics of relevance. Then we report the 2.5% quantile and the 97.5% quantile as the boundaries of our confidence interval with confidence level of $\alpha = 5\%$.

```
from random import choices
from sklearn.linear_model import Lasso
import warnings
warnings.filterwarnings('ignore')


y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')


#X = np.c_[np.ones(X.shape[0]), X, X**2, X**3, X**4]
X = np.c_[X, X**2, X**3, X**4]
x = np.arange(1, 12, 0.05).reshape((-1, 1))
#x = np.c_[np.ones(x.shape[0]), x, x**2, x**3, x**4]
x = np.c_[x, x**2, x**3, x**4]
indices = np.arange(0, X.shape[0])

drew = choices(indices, k=len(indices))

sampler = (choices(indices, k = len(indices)) for i in range(200))

CIS = np.percentile(np.array([Lasso(alpha=2, fit_intercept=True).fit(X[drew,:],␣
↪y[drew, :])\
                              .predict(x).tolist()
                              for drew in sampler]), [2.5, 97.5], axis = 0)
# x is 220 long
model = Lasso(alpha=2, fit_intercept=True)
model.fit(X, y)
y_hat = model.predict(x)
```

```
<matplotlib.legend.Legend at 0x7f305c0a5990>
```


lasso regression for polynome of 4th degree and $\lambda = 2$
lasso regression
Mean Prediction CI

CHAPTER

ELEVEN

EXTENSION: LOGISTIC REGRESSION AND THE GLM

There are other models that are relatives of the linear model that we discussed in this notebook. One of the most prominent is the **logistic regression**. This model belongs to the "**generalized** linear model" (GLM). The GLM may not be confounded with the "**general** linear model". The latter essentially expresses analysis of variance (ANOVA) in terms of linear regression. The **GLM** extends the linear regression beyond models with normal error distributions. This remark in the corresponding wiki-article is enlightening: read wikipedia for this

11.1 exponential family of distributions

From the perspective of modern statistics the GLM comprises many different linear models, among others the classical linear model. Every distribution in the exponential family can be written in the following form: $$f(y|\theta) = \exp\left(\frac{y\theta+b(\theta)}{\Phi}+c(y,\Phi)\right), where \theta is called the canonical parameter that in turn is a function of \mu, the mean. This function is ca
it is this function that linearizes the relation between the dependent and the independent variables. For the sake of completeness :b
\Phi)$$ is a function depending on the observation and the dispersion parameter.

11.1.1 Normal distribution

$$\begin{aligned} f(y|\mu,\sigma) = & \quad (2\pi\sigma^2)^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\frac{y^2-2y\mu+\mu^2}{\sigma^2}\right) \\ = & \quad \exp\left(\frac{y\mu-\frac{\mu^2}{2}}{\sigma^2}-\frac{1}{2}\left(\frac{y^2}{\sigma^2}+\log(2\pi\sigma^2)\right)\right), \quad \text{with} \end{aligned}$$

$\mu = \theta(\mu)$, i.e. μ is the canonical parameter and the link function is given by the identity function. Hence, the mean can be modeled directly without any transformation. The dispersion parameter Φ is given by σ^2, the variance. This case is the classical linear regression.

11.1.2 Poisson distribution

Now, for the Poisson distribution we have

$$\begin{aligned} f(y|\mu) = & \quad \frac{\mu^y e^{-\mu}}{y!} = \mu^y e^{-\mu}\frac{1}{y!} \\ = & \quad \exp\left(y\log(\mu)-\mu-\log(y!)\right), \quad \text{where} \end{aligned}$$

the link function is given by $\log(\mu)$. Note that the Poisson distribution does not have any dispersion parameter.

11.1.3 Bernoulli distribution ⇒ logistic regression

And finally the Bernoulli distribution from which we derive the logistic regression. Using the Bernoulli distribution, we can calculate the probabilities of experiments consisting of binary events. The classical example is coin flipping. Here, π is the probability of the coin showing 'head'; $(1-\pi)$ is the probability of the coin showing 'tail'. We can now calculate the probability of getting exactly 7 times head for 10 tosses with a fair coin: $\$\pi^7(1-\pi)^3 = 0.5^7 0.5^3 = 0.5^{10} = 0.0009765625\$$

Next, I demonstrate how we can rewrite the Bernoulli distribution to fit into the framework of the exponential family:

$$\begin{aligned} f(y|\pi) = \quad & \pi^y(1-\pi)^{1-y} = \exp\left(y\log(\pi) + (1-y)\log(1-\pi)\right) \\ = \quad & \exp\left(y\log(\pi) + \log(1-\pi) - y\log(1-\pi)\right) \\ = \quad & \exp\left(y\log(\tfrac{\pi}{1-\pi}) + \log(1-\pi)\right), \quad \text{where} \end{aligned}$$

the link function evaluates to $\log(\frac{\pi}{1-\pi})$. This function is also called the logit function whose reverse function is the logistic function. Hence, it is the logit that is modeled by a lineare function of the regressors: $\log(\frac{\pi}{1-\pi}) = a + b_1 x_1 + ... + b_j x_j$. If we plug the right hand term into the logistic function we get the estimated probabilities: $\$P(y=1|x) = \frac{\exp(a+b_1 x_1 + ... + b_j x_j)}{1+\exp(a+b_1 x_1 + ... + b_j x_j)}.\$$

Here, I showed that the classical linear regression with normal error terms can be seen as a special case of a much wider family of models comprising all distributions out of the exponential family. (For a more complete treatment of other distributions see again https://en.wikipedia.org/wiki/Generalized_linear_model.)

CHAPTER

TWELVE

GLMNET

In the statistical language R, there exists a library called 'glmnet'. This package implements the elastic net as we discussed here but for the glm and not only for the classical linear regession. https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html

There exists also a python package implementing glmnet by using the exact same fortran code: **glmnet-python**. There are some subtleties in the implementation that are different from the elastic net version as provided by sklearn. https://pypi.org/project/glmnet-python/

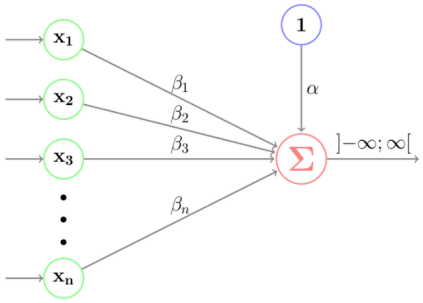
CHAPTER

THIRTEEN

NEURAL NETWORK

We can also cast linear regression into a neural network context. The network has no hidden layer. The activation function in the output neuron is either the identity function $y = x$ for classical linear regression or the logistic function for logistic regression.

classical linear regression



Remember, we included the intercept α into the vector β by including an all-ones vector into the matrix $\mathbf{X}$. The equation is hence written:

$$\mathbf{y} = \mathbf{X}\beta$$

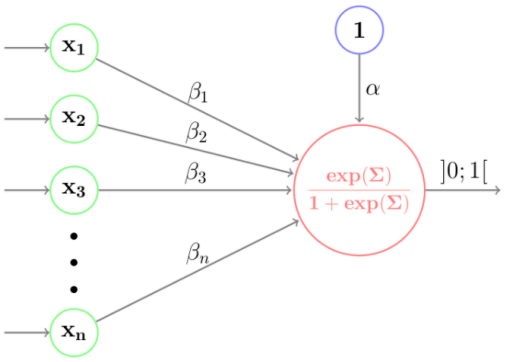
In neurall network context, the vector β is called the network weights and often is denotedn as $\mathbf{W}$.

13.1 classical linear regression

Remember, we included the intercept α into the vector ? by including an all-ones vector into the matrix $\mathbf{X}$. The equation is hence written: $\mathbf{y} = \mathbf{X}?Inneurallnetworkcontext, thevector\mathbf{\beta}iscalledthenetworkweightsandoftenisdenotednas\mathbf{W}$.

Why are the weight-vectors $\mathbf{W}$ in upper-case?In a regression-context, we usually use lower-case letters like ? or $\mathbf{b}$?

logistic regression



For logistic regression, the activation function is changed. Now, it is not the identity function, but the logistic function:

$$P(y = 1|x) = \frac{\exp(a + b_1x_1 + \ldots + b_jx_j)}{1 + \exp(a + b_1x_1 + \ldots + b_jx_j)}$$

This function approaches 0, 1 asymptotically.

13.2 logistic regression

For logistic regression, the activation function is changed. Now, it is not the identity function, but the logistic function: $$P(y = 1|x) = \frac{\exp(a+b_1x_1+\ldots+b_jx_j)}{1+\exp(a+b_1x_1+\ldots+b_jx_j)}$$ This function approaches 0, 1 asymptotically.

13.2.1 Weight decay

In the neural network literature, the l_2-penalty term is called "weight decay". It is not a parameter of the single layers or neurons, but of the optimizer. As with regularized regression, the weight decay is written: $L' = L + \lambda \sum_i w_i^2, where Listheactuallossandw\_i$ are the weights of the incoming connections of a neuron.