My sample book

The Jupyter Book Community

Jul 13, 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

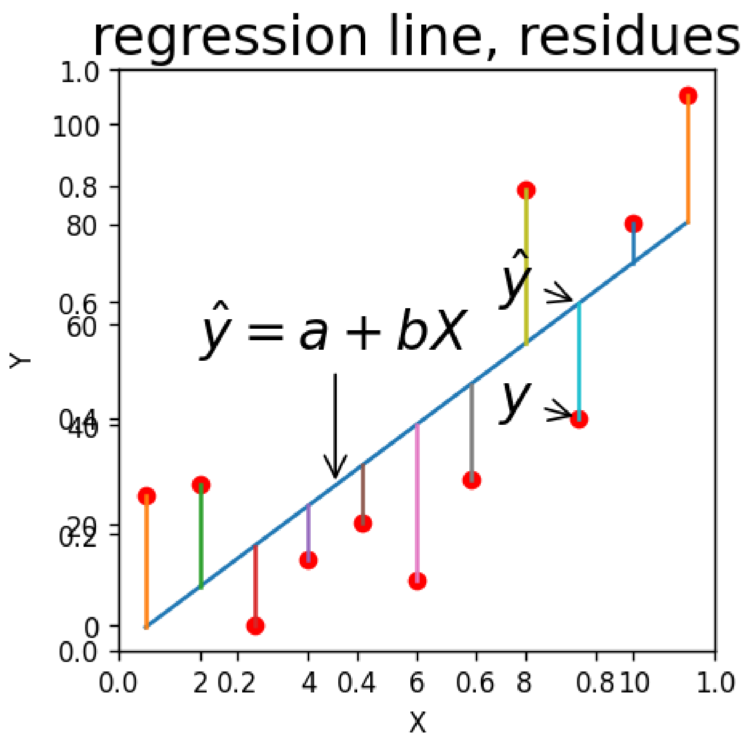
LINEARE REGRESSION

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 $\hat{y}$.

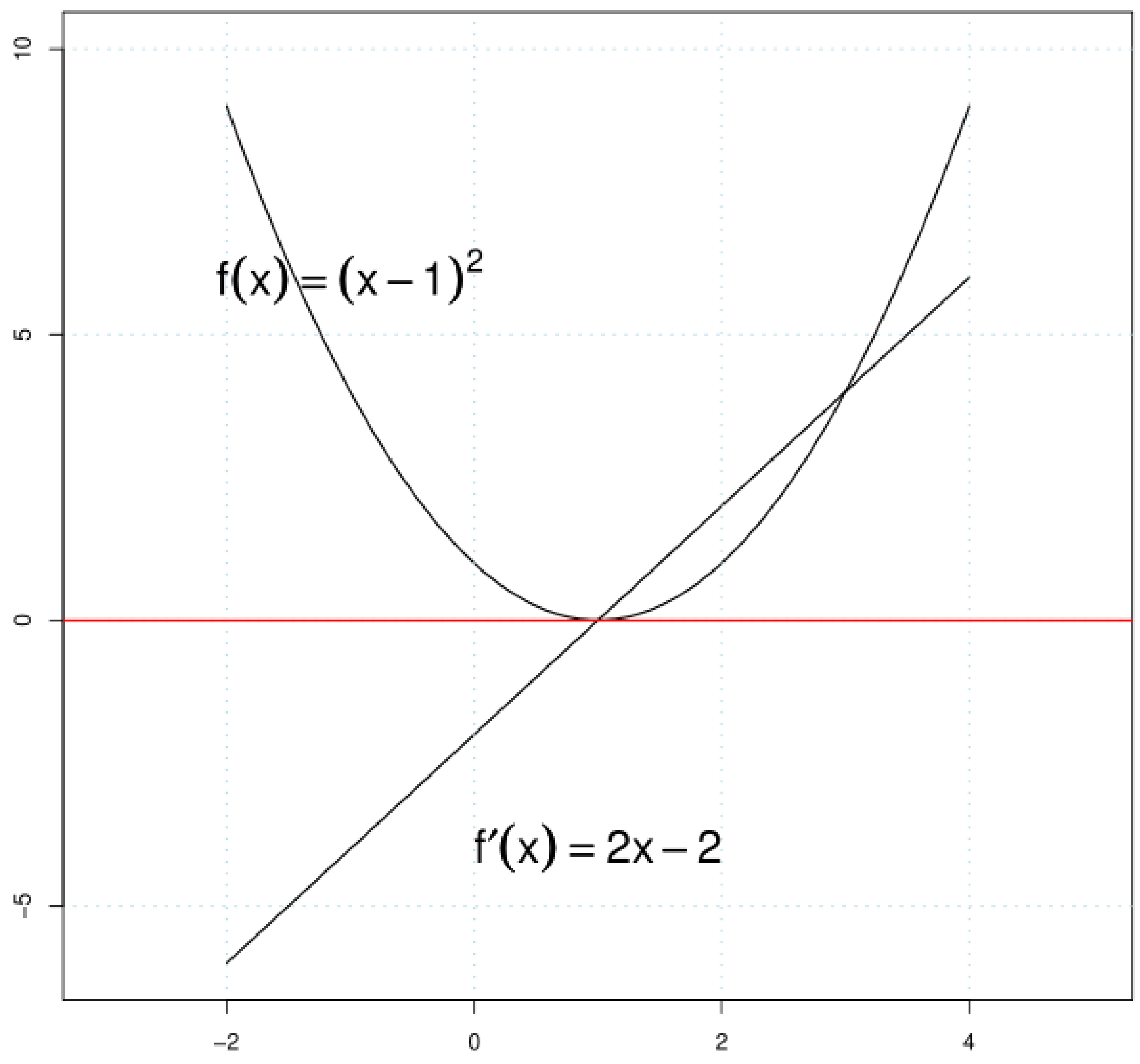
CHAPTER

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.

CHAPTER

THREE

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 empirical b parameter is: {b}')
print(f'our empiricial 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 empirical b parameter is: 8.079124453577005
our empircial a parameter is: -8.490321540681798
```

CHAPTER

FOUR

MULTIVARIATE CASE: MORE THAN ONE X VARIABLE

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

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

Next, we can include the constant term a into the vector b. This is done by adding an all-ones column to $\mathbf{X}$:

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

In matrix notation this is written: \$$\mathbf{y} = \mathbf{Xb}$\$

CHAPTER

FIVE

DERIVATION OF B FOR THE MATRIX NOTATION

We apply the same steps as for the derivation above:

- 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**

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

For the last step, we need the inverse of a matrix: $(\mathbf{X}'\mathbf{X})^{-1}$

POLYNOMIAL REGRESSION AS AN EXAMPLE FOR MORE THAN ONE VARIABLE

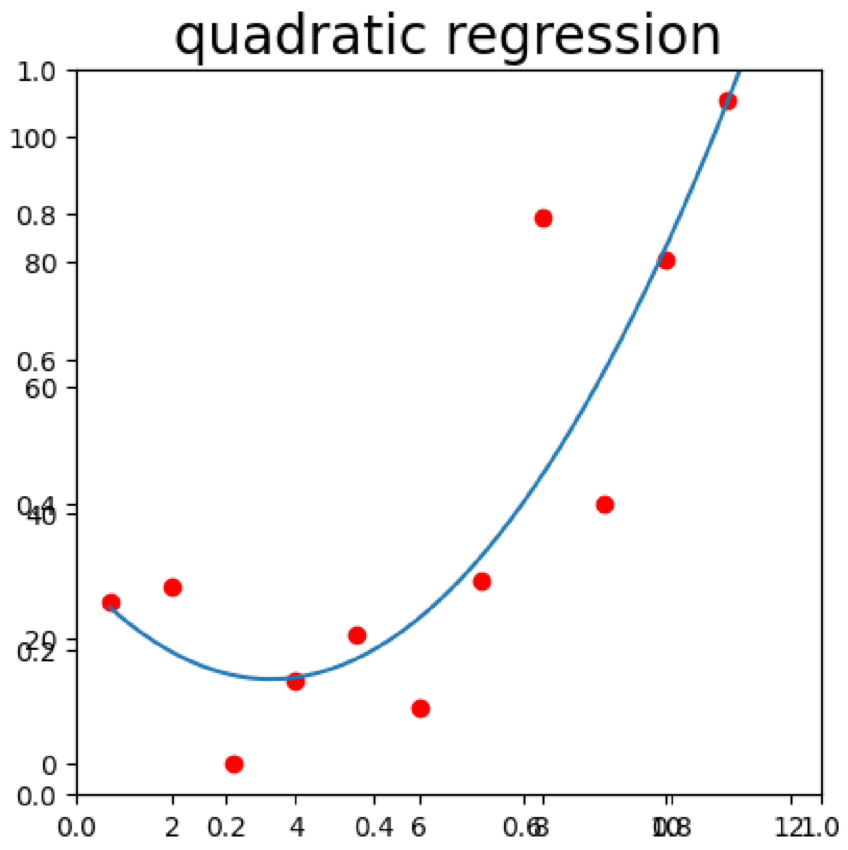
In order to easily demonstrate multiple linear regression, we can derive a new variable out of the variable x. For example we could take log(x) or – as done here – take the square of it x^2 (the quadratic term): $y = a + b_1 x + b_2 x^2$

Some important points are:

- we now have two variables, i.e. we can apply our formula for matrix notation
- more variables will probably lead to a better fit
- the resulting regression line is not a straight line. **The term "linear" in linear regression signifies that the equation is linear in its parameters a, b_1, b_2. It does not mean that the regression line has to be a straight linear line!!**

```
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)
```



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

6.1 Overfitting

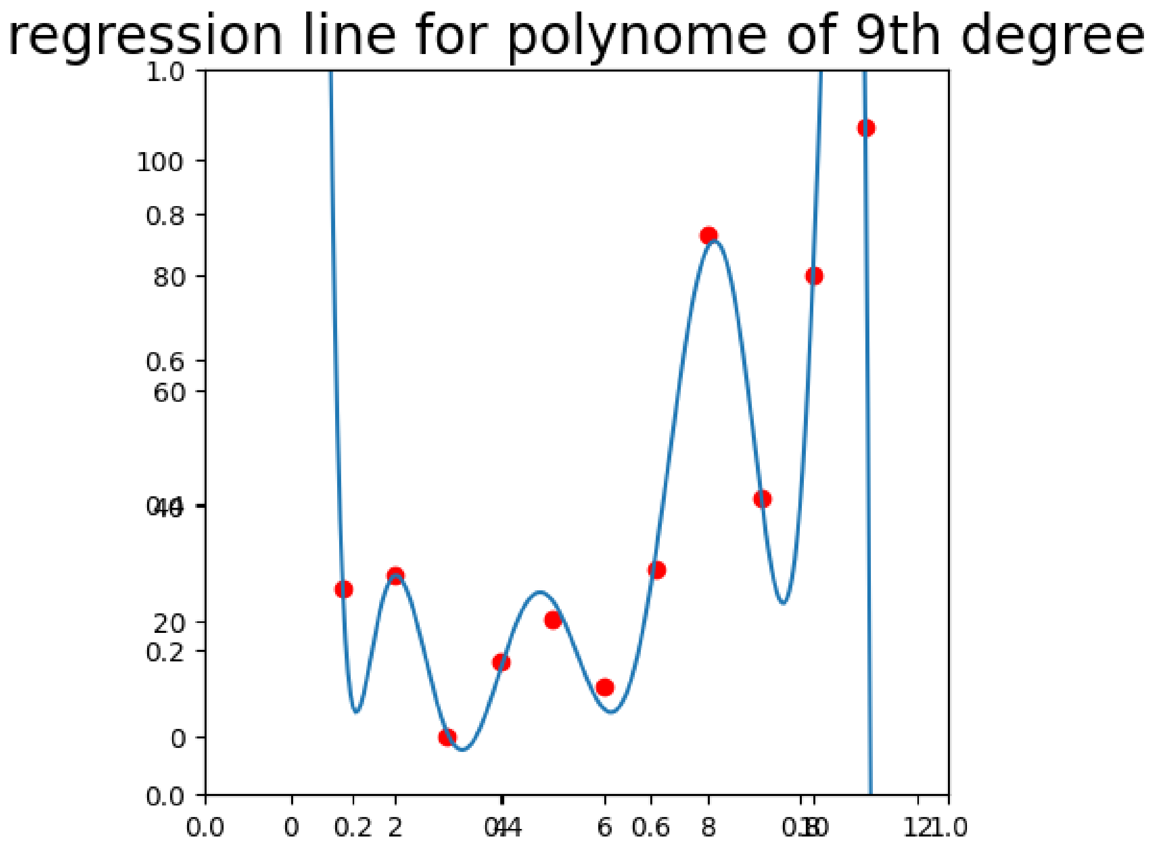
We continue with adding variables and exagerate a little bit

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)
```



6.2 perfect fit: as many variables as data samples

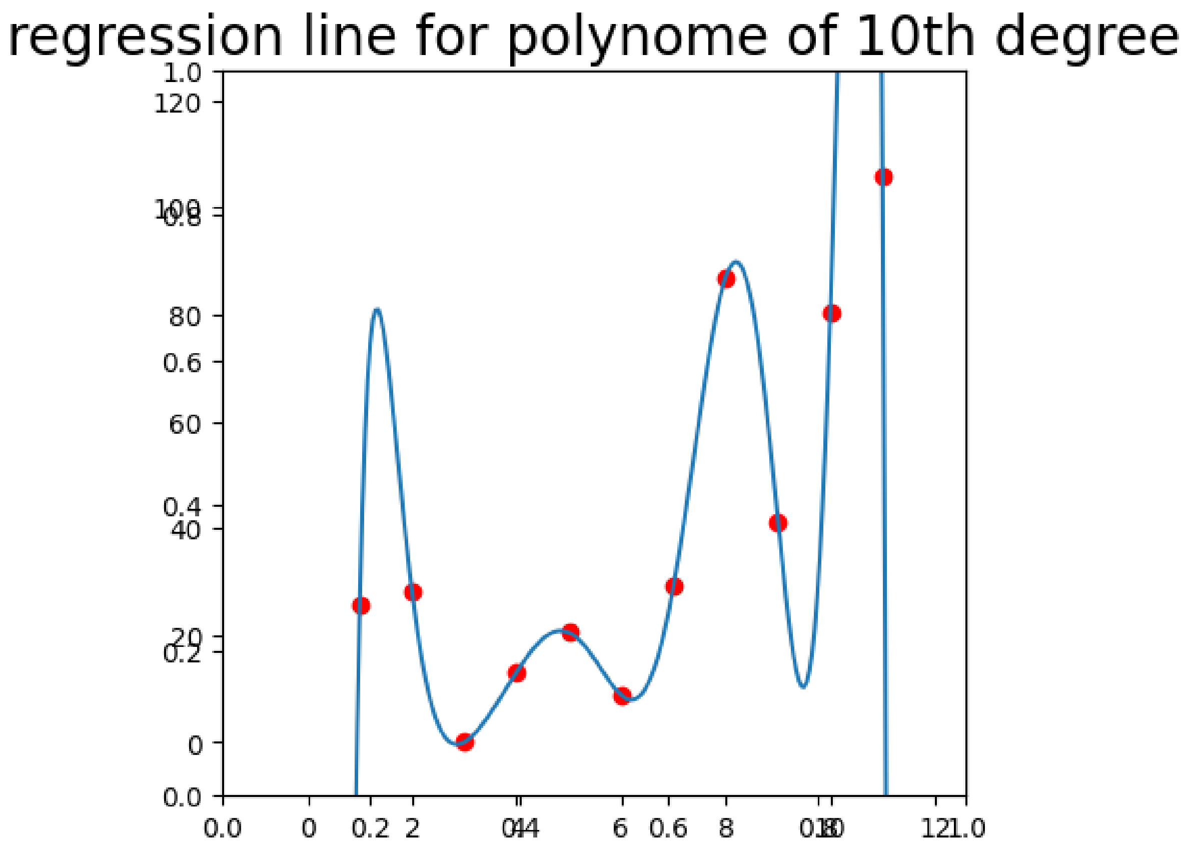
A perfect fit is possible as is demonstrated next. We have as many variables (terms derived from x) as observations (data points). So for each data point we have a variable to accommodate it. **Note**, that a perfect fit is achieved with 10 variables + intercept. The intercept is also a parameter and 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

SEVEN

WHAT HAPPENS IF WE HAVE MORE VARIABLES THAN DATA POINTS?

The short answer: a unique solution is not possible because there are infinitifely many possible ways to adjust p parameters to accommodate n observations when $p > n$.

The long answer: inversion of the matrix $\mathbf{X}'\mathbf{X}$ is not possible.

However, there are decompositions for matrix inversions that allow to invert singular matrices. Numpy is using such a decomposition, called 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 with 1s in the diagonale and zeros otherwise: \$$\mathbf{I} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$\$

```
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
```

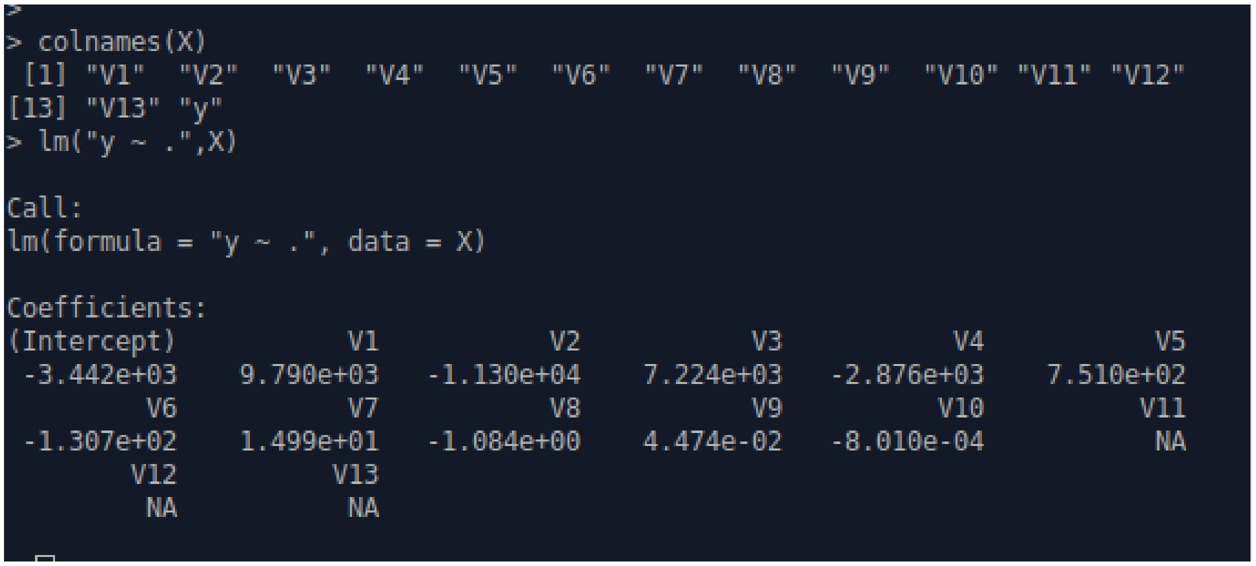
CHAPTER

EIGHT

STATISTICAL PACKAGE R

The R statistical package is behaving still in another way. No warning is issued but coefficients are only computed for 11 variables (intercept included).

```
/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

NINE

DEALING WITH OVERFITTING

As we could see, if there are many variables and only few observations, classical linear regression tends to overfit heavily. One solution for this problem is to shrink the coefficients $b_1, b_2, b_3, ...$. This can be achieved by making the error bigger with bigger coefficients. The algorithm strives to reduce the error and hence has to shrink the coefficients to lower values.

9.1 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.

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:

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

Interestingly, this error term has a closed form solution, i.e. there is a analytical solution and we do not have to resort to iterative algorithms. However, remember that we included the intercept parameter a in **b** and added an extra column with ones to the matrix **X**. By computing $\lambda\mathbf{b}'\mathbf{b}$ we would also shrink the intercept parameter - what is not meaningfull since its effect is to account for the means of the variables: $a = \bar{y} - b\bar{x}$ In order to remove this term, we have to center the variables. When $\bar{y} = 0$ and $\bar{x} = 0$ then a vanishes. The solution for **b** is then given as: $\$\hat{\mathbf{b}} = (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}'\mathbf{y}\$$

Following Hastie et al., originally, this was introduced to cope the rank deficency problems. When the algorithm was first proposed, it was not possible to invert a square matrix not of full rank. Hence, adding a small positive amount to its diagonal solved this problem. This can be demonstrated with our numerical example:

```
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))}')
```

(continues on next page)

(continued from previous page)

```
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])
```

9.1.1 example of ridge regression

Next, we will apply ridge regression as implemented in the python sklearn library and compare the results to the analytical 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

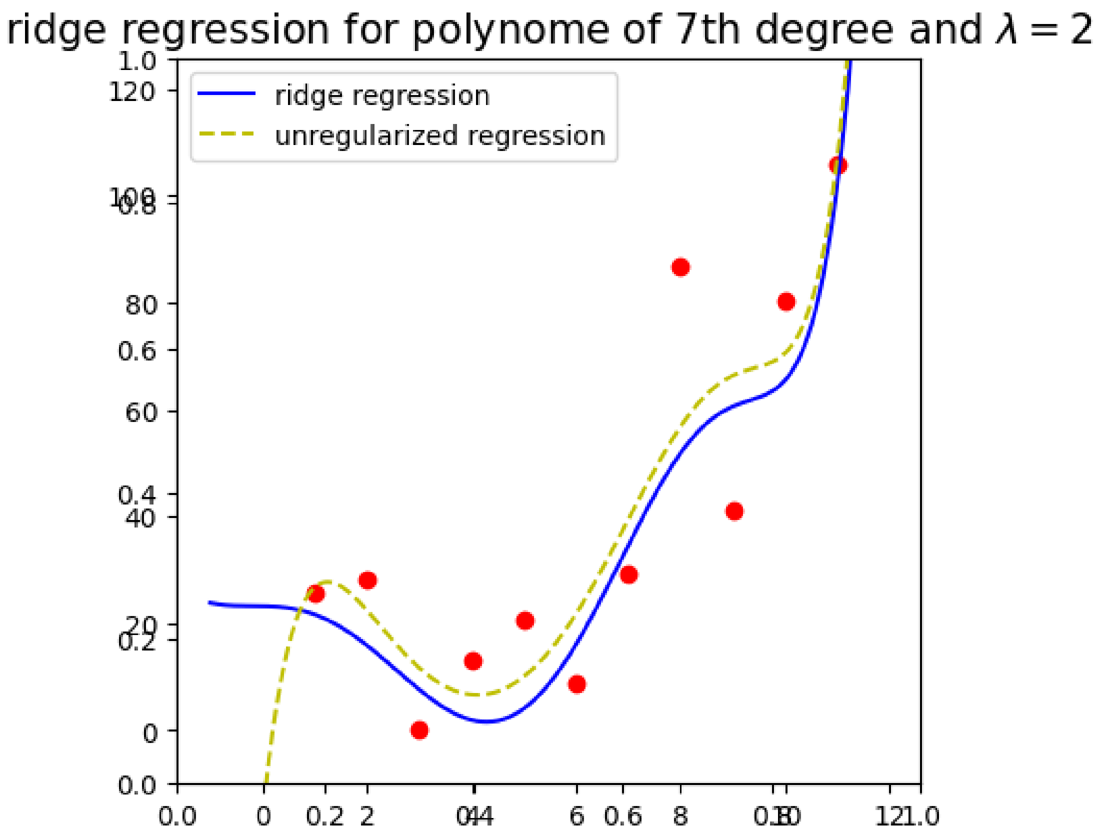
```
from sklearn.linear_model import Ridge
y = np.load('/home/martin/python/fhnw_lecture/scripts/regression_y.pickle.npy')
X = np.load('/home/martin/python/fhnw_lecture/scripts/regression_X.pickle.npy')

X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7]
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'
```

(continues on next page)

(continued from previous page)

```
      + 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]]
```



9.2 Lasso

To make the error term bigger, alternatively, we could add the absolute value $\lambda \cdot |b|$ to the error:

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

For two variables we can write:

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

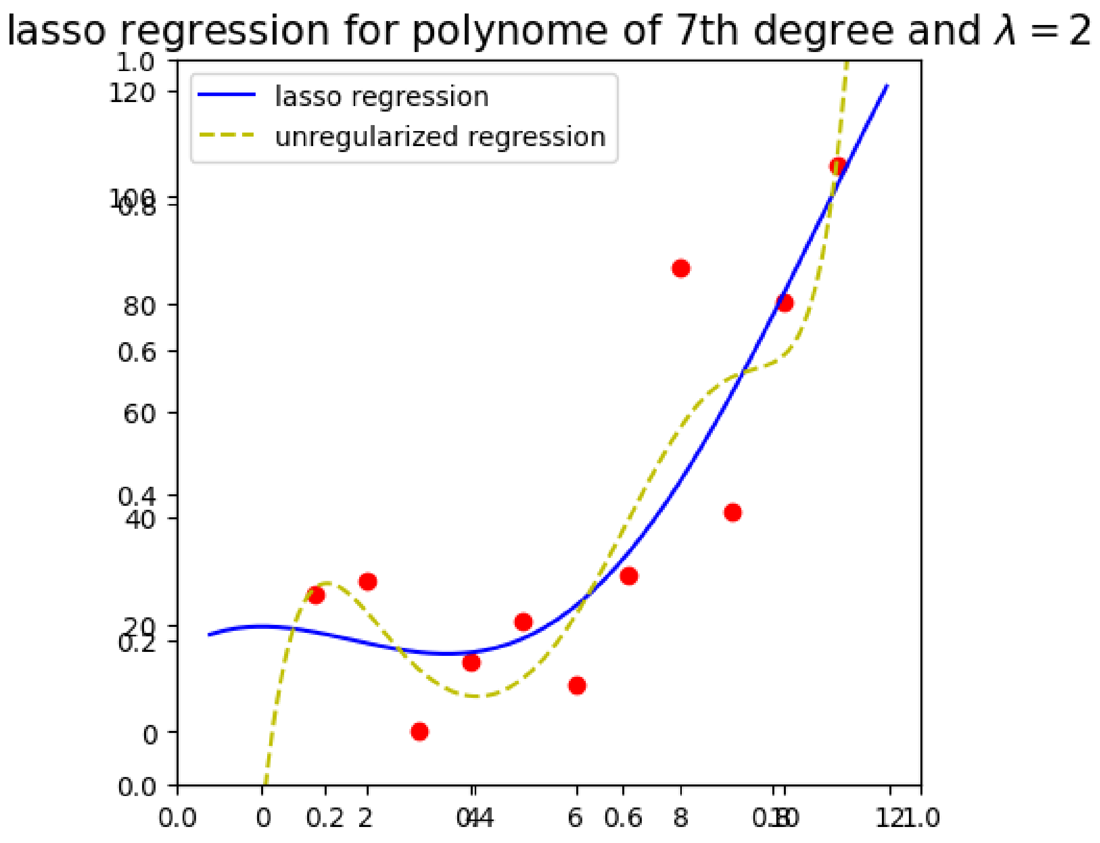
Unfortunately, and contrarily to ridge regression, there exists no closed form expression for computing the coefficients for the lasso.

9.2.1 lasso 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]
```

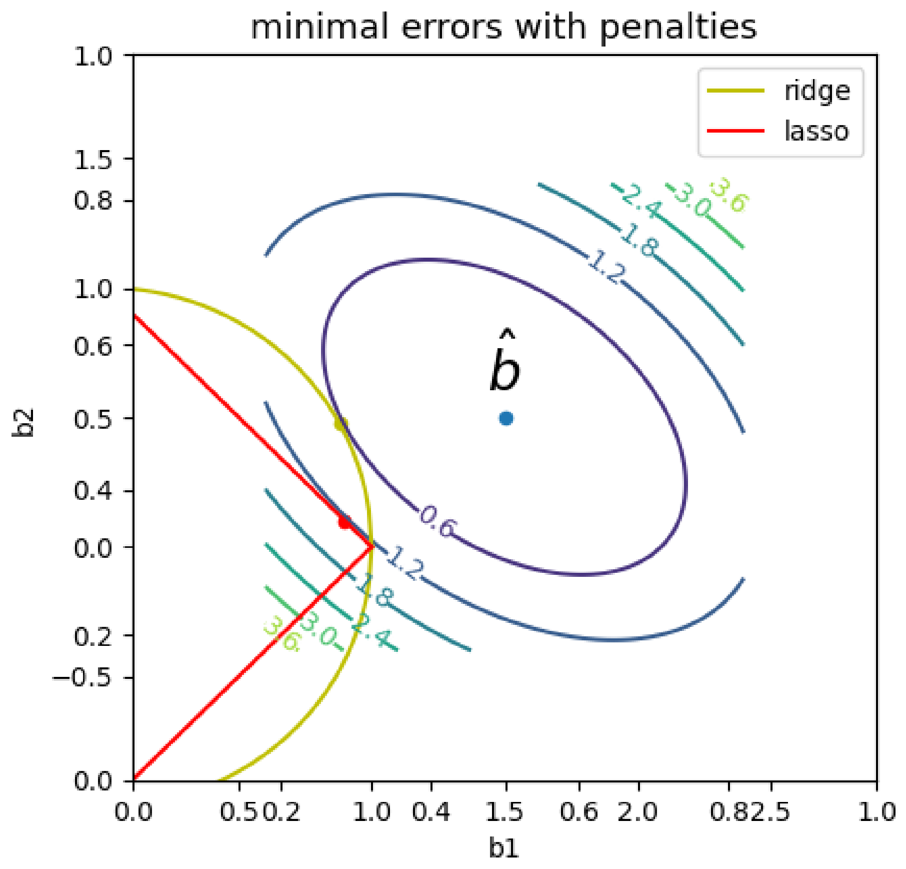


9.3 the difference between ridge and lasso

Ridge regression tends to shrink all parameters in an equal manner. Lasso often leads to solutions with some parameters converged to zero. Hence lasso can also be a variable selection algorithm. Compare the results for our little toy example. Essentially, lasso has only two parameters (for X^2 and X^3) that are different from zero. All other parameters' absolute values are smaller than 0.0007. To see why lasso shrinks parameters to zero, we will:

- generate a random dataset $y = 1.5 \cdot x_1 + 0.5 \cdot x_2$
- compute mean squared error (MSE) for a grid of different values for b_1 and b_2
- plot error contour
- show the geometric shape of the penalties: $\lambda \sum_j b_j^2$ and $\lambda \sum_j |b_j|$
- indicate the optimal point for the combination of penalty and MSE-error

```
the model parameters from data generation could be recovered: [1.5 0.5]
```



```
optimal coefficients of the ridge solution: 0.8787878787878789 and 0.47721259842495806
optimal coefficients of the lasso solution: 0.8989898989898991 and 0.10101010101010088
```

In the figure above the optimal solution (without regularization) is at (1.5, 0.5) and is indicated by $\hat{b}$. This solution is associated with 0 mean squared error (MSE). The contour lines indicate the mse-error corresponding to the respective choice of the parameter values. The yellow ball like line and the red triangular line indicate parameter values corresponding to a penalty budget of 1 for the ridge and the lasso respectively. The best solution is given by the point on the line associated with the least mse-error. These points are also indicated. As can be seen, the ridge regression shrinks both parameters whereas the lasso drives the b_2 estimate towards zero and the b_1 estimate towards one.

CHAPTER

TEN

ELASTIC NET

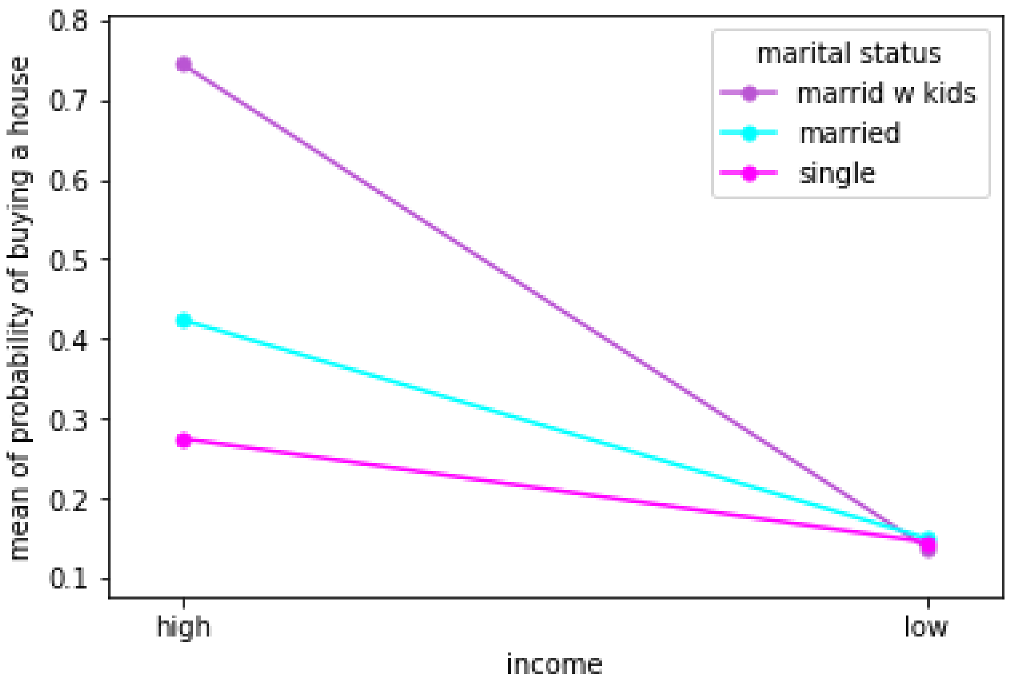
The ridge regression and the lasso tend to shrink parameters quiet differently. A compromise that gaines huge popularity is a combination of the two: $ $\lambda \sum_j(\alpha b_j^2 + (1 - \alpha)|b_j|) Besidesλwhichgearstheamountofregularization, anewparameter,αweightsthecontributionofthe$l_2 penalt
penalty due to lasso regression. We will use elastic net in our data example.

CHAPTER

ELEVEN

INTERACTION

If the effect of one variable on the outcome depends on the value of another variable. As an example we could try to model the probability that someone is going to buy a house. A very important variable for buying a house will be the monthly income. Another one could be the marital status (single, maried, maried with kids). Maried persons with kids will be very inclined to buy a house if the income situation is favorable. Singles, even with high income will not be considering buying a house. So, the effect of income is different for the levels of marital status:



This introducing example comprised categorical variables. Interaction effects may also exist for continuous variables. In this case it is just harder to visualize. Again, we will construct our own data example and build a strong interaction into it. To properly visualize the data, we have to put one variable into bins. However, the scatter plot allows for an intuitive understanding of the interaction of two continuous variables.

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-25-32891ff7e5e0> in <module>
----> 1 import seaborn as sns
      2 n = 500
      3 x = np.random.uniform(size=n)
      4 m = np.random.normal(loc = 0.5, scale = 1, size = n)
      5
```

(continues on next page)

(continued from previous page)

```
ModuleNotFoundError: No module named 'seaborn'
```

As can be seen, the interaction effect is build into the example data, by simply multiplying two variables. Lets see, if we can recover the coefficients. We will also see that the model fit is better when the interaction term is included:

```
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')
```

11.1 some considerations

Let's assume, we have a data set with 70 different variables. Since we do not know anything about the relationsship of the variables to the dependent variable (y), nor of the variables among each other, we are inclined to construct a lot of new variables:

- we can add all 70 quadratic terms (x_j^2)
- we can add all 70 cubic terms (x_j^3)
- we could add all $\binom{70}{2} = 2415$ first-order interactions among the 70 variables
- instead, we could add all first-order interactions among the 210 variables including the quadratic and the cubic terms: $\binom{210}{2} = 21945$
- besides quadratic and cubic transformations, there my be other transformations leading to better results, like the log-transform.

As you can see, the number of possible variables can grow very fast, when considering all possible effects that might be present in the data. Sometimes, there exists second-order interaction effects, that were not mentioned in the considerations above. All these variables would likely lead to severe overfitting if we would naively include them in our linear regression model. **That's why we introduced regularization techniques like the elastic net and its components, the ridge and the lasso**.

CHAPTER

TWELVE

HOW CONFIDENT ARE WE ABOUT OUR PREDICTIONS

Essentially, there are two questions that one could ask after fitting a linear model:

- How confident are we about the estimated parameters **b**? This is most often asked by statisticians because for them, the interpretation of the coefficients is of paramount interest.
- How confident are we about the predictions? This is asked in machine learning because we want to apply the model to unseen data and use the predictions in our business process. Here, we have to deal with two different questions:
  - How much will the mean response, our prediction - the regression line - vary?
  - How much variation is in the observations y given the level of X?

12.1 Recap of assumptions underlying regression

- **Linearity**: The regression line is a good fit for the relation between **X** and **y**, i.e., if there is a quadratic trend in the data and we fit a model without quadratic term, the assumptions are not met (–> **bias**).
- **Homoscedasticity**: The variance of the residuals is identical for all values of **X**.
- **Normality**: The values of **y** given a certain **x**, i.e. of **y**|**x** are normaly distributed.

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$are the residuals, i.e., the differ$ $y_i \pm t_{1-\alpha/2,N-p} \cdot \hat{\sigma}_e. Here, t$\{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']]
```

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())
```

12.2 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)
```

CHAPTER

THIRTEEN

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

13.1 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), whereθiscalledthecanonicalparameterthatinturnisafunctionofμ, themean.Thisfunctioniscal
$itisthisfunctionthatlinearizestherelationbetweenthedependentandtheindependentvariables.Forthesakeofcompleteness$:b
\Phi)\$ is a function depending on the observation and the dispersion parameter.

13.1.1 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.

13.1.2 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.

13.1.3 Bernoulli distribution $\Rightarrow$ 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: $\log(\frac{\pi}{1-\pi}) = a + b_1x_1 + ... + b_jx_j$. If we plug the right hand term into the logistic function we get the estimated probabilities: \$$P(y=1|x) = \frac{\exp(a+b_1x_1+...+b_jx_j)}{1+\exp(a+b_1x_1+...+b_jx_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

FOURTEEN

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

FIFTEEN

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.

15.1 classical linear regression

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

Why are the weight-vectors **W** in upper-case?In a regression-context, we usually use lower-case letters like ⍰ or **b**?

15.2 logistic regression

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

15.2.1 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$L$istheactuallossandw_i are the weights of the incoming connections of a neuron.