
1st Session

Martin Biehler

Feb 21, 2023

CONTENTS

I	intro	3
1	Overview	5
1.1	Knowledge Discovery -> Data Mining -> Data Science -> Machine Learning	6
II	All about regression	7
2	Linear Regression	9
2.1	Analytische Herleitung der Parameter der Linearen Regression	10
2.2	multivariate case: more than one x variable	13
2.3	Polynomial regression as an example for more than one variable	14
2.4	What happens if we have more variables than data points?	18
2.5	Overfitting and the Bias-Variance Tradeoff	20
3	Dealing with overfitting	23
3.1	Ridge regression	23
3.2	Lasso	26
3.3	the difference between ridge and lasso	29
3.4	ElasticNet	30
4	Interaction	31
4.1	some considerations	33
5	Wie zuversichtlich sind wir hinsichtlich unserer Modell-Vorhersagen	35
5.1	Recap of assumptions underlying regression	35
5.2	Bootstrap	40
6	Extension: logistic regression and the GLM	43
6.1	exponential family of distributions	43
7	Neural Network	45
7.1	classical linear regression	46
7.2	logistic regression	47
III	Cross Validation, Stacking and Mean Encoding	49
8	Cross-Validation	51
8.1	example in python:	52
9	Nested cross-validation	53

10 Stacking	55
11 Mean-encoding or target encoding:	57
11.1 only for classification	57
11.2 Some Literature	58
 IV Data Leakage and Data Dependence	 59
12 Date Leakage and Dependent Data	61
13 Sources of data leakage	63
13.1 train data contains features that are not available in production	63
13.2 future data somehow slipped into the training set	63
13.3 there is one feature that interacts with the target	64
13.4 Some more cases where we have data leakage:	64
13.5 credit card applications	65
13.6 Solution:	70
 14 Dependency between data-samples	 71
14.1 First contact with Pipeline	71
14.2 the correct way to do it: oversample within each fold:	74
14.3 Some more cases where we have dependent data	74

Der Input diese Skripts ist **in** der Vorbereitung eines Seminars entstanden, welches ich
im Sommersemester 2021 zum ersten Mal an der FHNW gegeben habe.

Part I

intro

OVERVIEW

1. Session

- Regression
- Cross-Validation, Stacking, Mean-Encoding
- Data Leakage, Dependent Data

2. Session

- Tree Methods
- Variable Importance
- Imbalanced Data

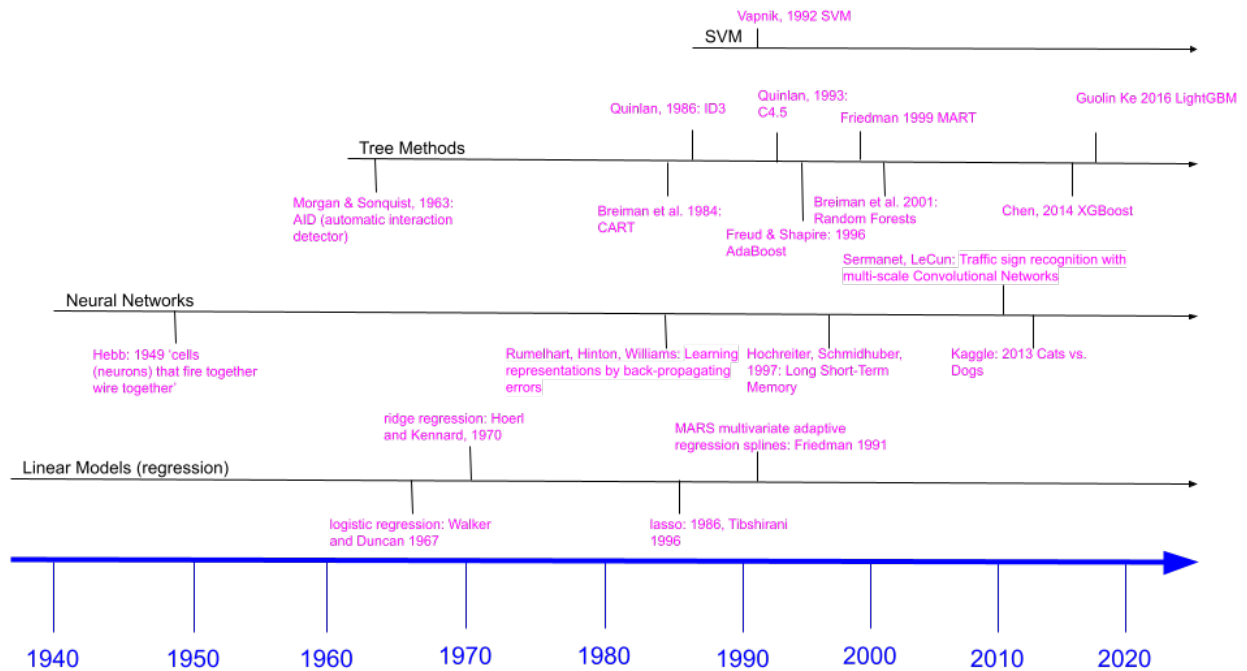
3. Session

- Kernel Methods
- AutoML
- Clustering
- Questions regarding the examen
- Questions regarding the Abschlussarbeit

4. Session

- discussion of the results of the Abschlussarbeit
- how to use pretrained Neural Networks in the data-science process
- Q&A

1.1 Knowledge Discovery -> Data Mining -> Data Science -> Machine Learning



Part II

All about regression

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 Klasse `LinearRegression` aus `sklearn.linear_model` gerechnet. Die Vorhersage (d.h. die Geradengleichung) ergibt sich aus den Koeffizienten durch

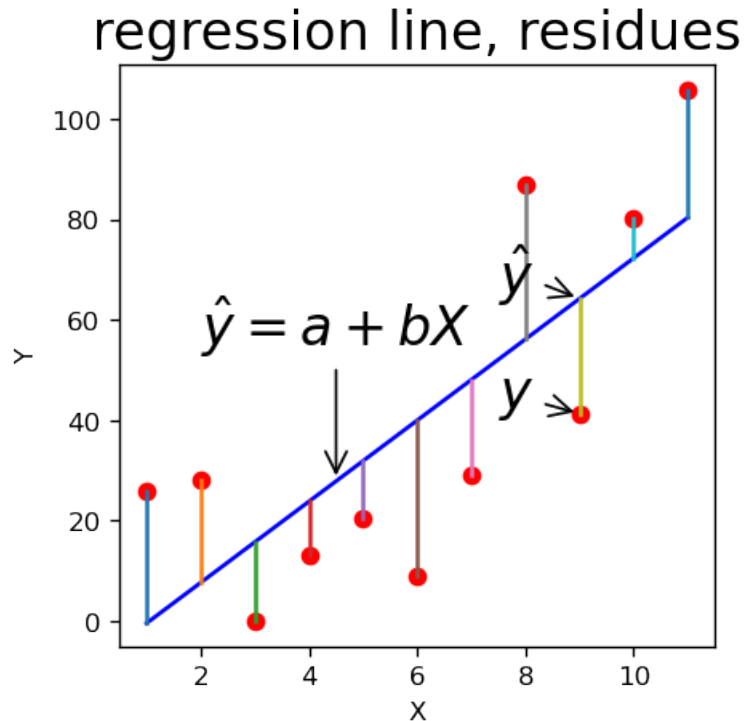
$$y = a + bX$$

```
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.

```
/home/martin/miniconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3:
↳MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous
↳axes currently reuses the earlier instance. In a future version, a new instance
↳will always be created and returned. Meanwhile, this warning can be suppressed,
↳and the future behavior ensured, by passing a unique label to each axes instance.
This is separate from the ipykernel package so we can avoid doing imports until
```



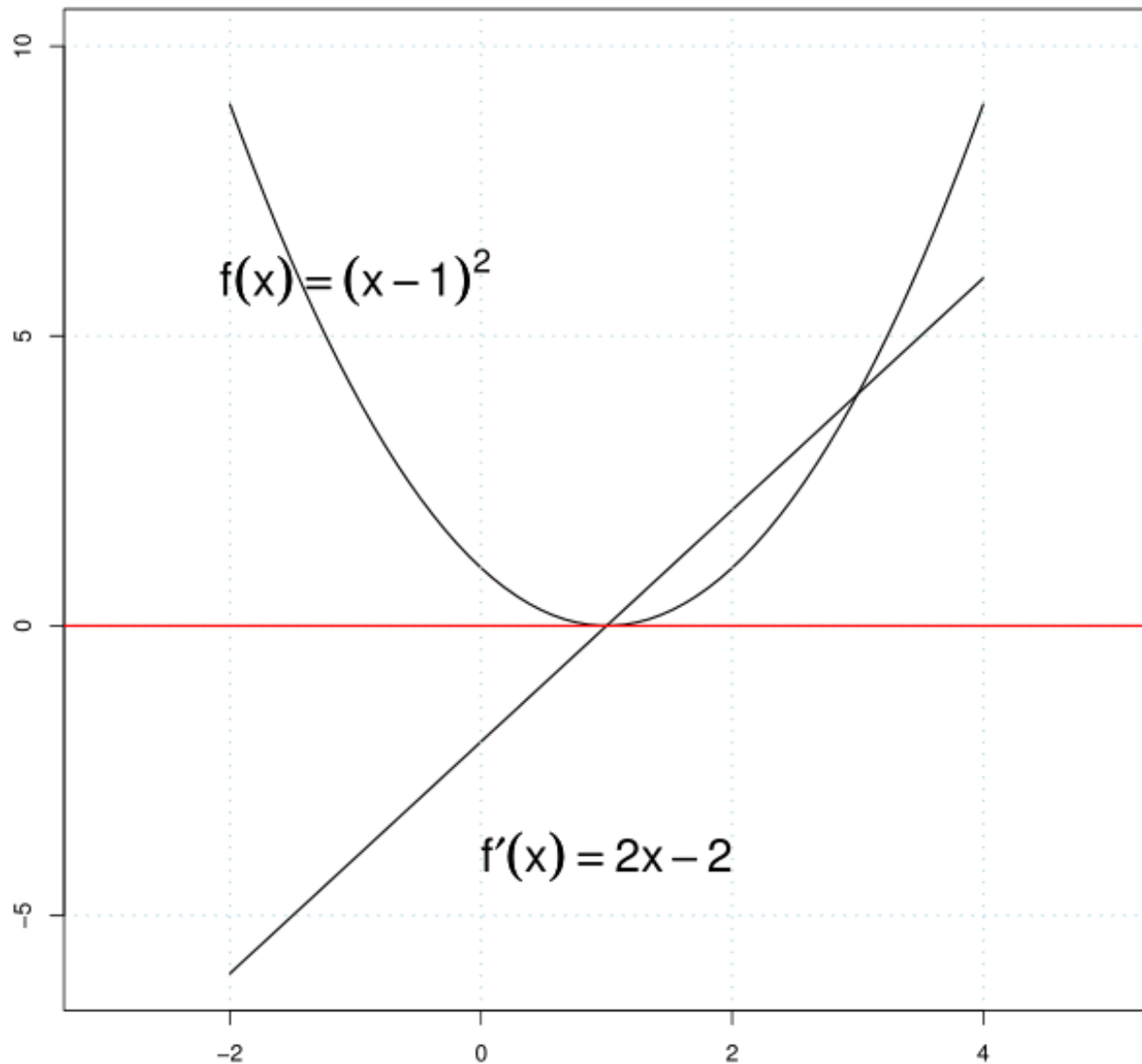
Residuen

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} .

2.1 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 oft 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.

Optimization I: univariate case

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):

$$\frac{\delta \sum_i^n (y_i - \hat{y}_i)^2}{\delta a} = -2 \sum_i^n y_i + 2b \sum_i^n x_i + 2na = 0$$

$$2na = 2 \sum_i^n y_i - 2b \sum_i^n x_i$$

$$a = \frac{2 \sum_i^n y_i}{2n} - \frac{2b \sum_i^n x_i}{2n}$$

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

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

Optimization I: univariate case

- 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 Formel: 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 `LinearRegression` 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: 8.079124453577007
our self-computed a parameter is: -8.490321540681805
```


2.2 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.

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

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

Den konstanten intercept Term (a) können wir mit in den Vektor der Parameter \mathbf{b} aufnehmen, indem wir in \mathbf{X} eine Einserspalte hinzufügen. Somit wird die Schreibweise sehr kompakt und der intercept a wird nicht mehr explizit aufgeführt:

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

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

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.

Optimization I: multivariate case

- 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 \mathbf{b}
- we set the result equal to zero and solve for \mathbf{b}

Optimiztaion I: multivariate case

$$\frac{\delta}{\delta \mathbf{b}} = -2\mathbf{X}'\mathbf{y} + 2\mathbf{X}'\mathbf{X}\mathbf{b} = 0$$

$$2\mathbf{X}'\mathbf{X}\mathbf{b} = 2\mathbf{X}'\mathbf{y}$$

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

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.

2.3 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 univariaten Variable x . Das Praktische ist, dass sich der Sachverhalt der Multivariaten Regression noch immer sehr schön 2-dimensional darstellen lässt.

$$y = a + b_1x + b_2x^2$$

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, b_1, b_2 ist. Für alle Werte einer Variablen x_1 gilt der gleiche Parameter b_1 . Es bedeutet nicht, dass die Regressions-Funktion durch eine gerade Linie gegeben ist!**
- ausserdem bedienen wir uns hier eines Tricks: Die Variable x^2 müsste eigentlich eine eigene Achse bekommen. Dann wäre die Regressions-Gerade wieder eine gerade Linie - nur lässt sich das leider nicht mehr schön darstellen.

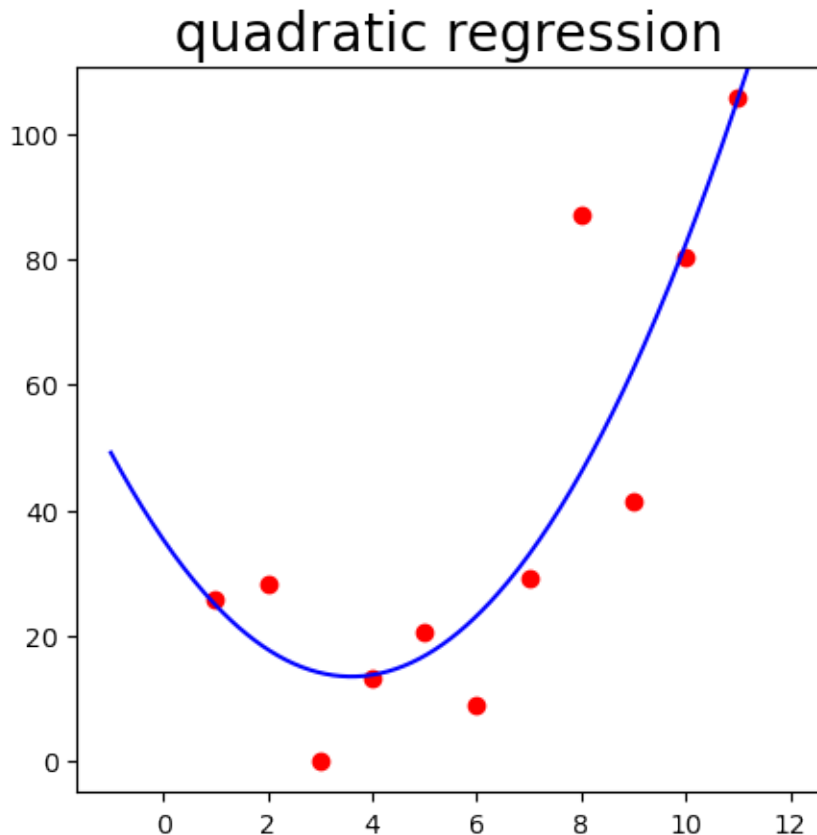
Nachfolgend fügen wir die weitere Variable durch Quadrieren der bisherigen Variable hinzu und berechnen abermals das Lineare Modell aus `sklearn.linear_model`.

```
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 intercept 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.

- `.T` transponiert eine Matrix
- `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_}')
```

(continues on next page)

(continued from previous page)

```
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]]
```

2.3.1 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. Man sagt dann, das Model “**generalisiert**” sehr schlecht. 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 genau so viele Paramter (10 Steigungskoeffizienten + intercept) hat wie Daten-Messpunkte.

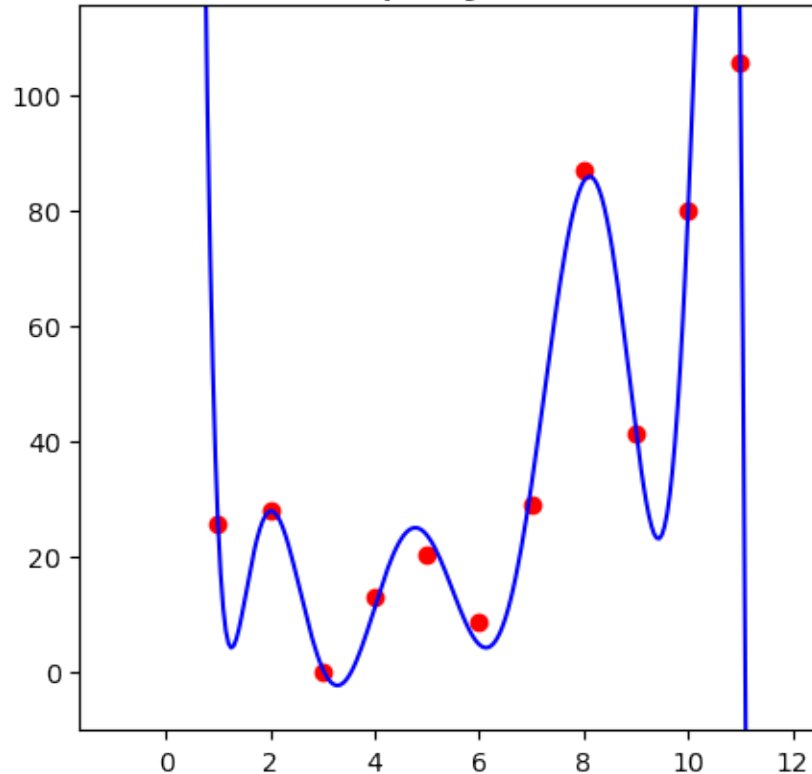
The important points to note here

- the fit to our empirical y-values gets better
- at the same time, the regression line starts behaving strangely
- 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)
```

regression line for polynome of 9th degree



2.3.2 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.
 1.3028575e+04  7.22272630e+03 -2.87529040e+03
```

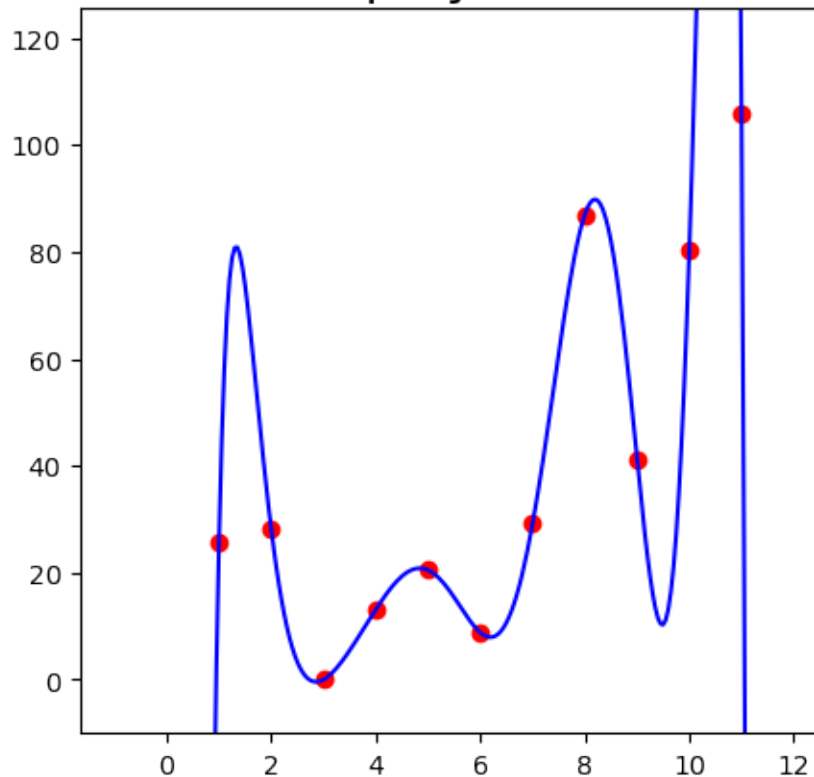
(continues on next page)

(continued from previous page)

```
7.50863939e+02 -1.30675765e+02 1.49834150e+01 -1.08409478e+00
4.47395935e-02 -8.00879370e-04]]
```

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

regression line for polynome of 10th degree



2.4 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 $X'X$ nicht. Mittlerweile gibt es Näherungsverfahren, die dennoch Ergebnisse liefern - wenn auch sehr Ungenau.

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 matrix S that solves $(X'X)^{-1}S = I$. I is called the eye-matrix with 1s in the diagonale and zeros otherwise:

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

Die entscheidende Zeile im nachfolgenden Code ist:

```
S = solve(inv(np.dot(X.T, X)), np.eye(13))
```

Sie besagt: gib mir die Matrix S , die multipliziert mit $(X'X)^{-1}$ die Matrix 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/imbalanced/lib/python3.7/site-packages/ipykernel_
↪launcher.py:15: LinAlgWarning: Ill-conditioned matrix (rcond=3.85212e-21): result
↪may not be accurate.
from ipykernel import kernelapp as app
```

2.4.1 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.

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

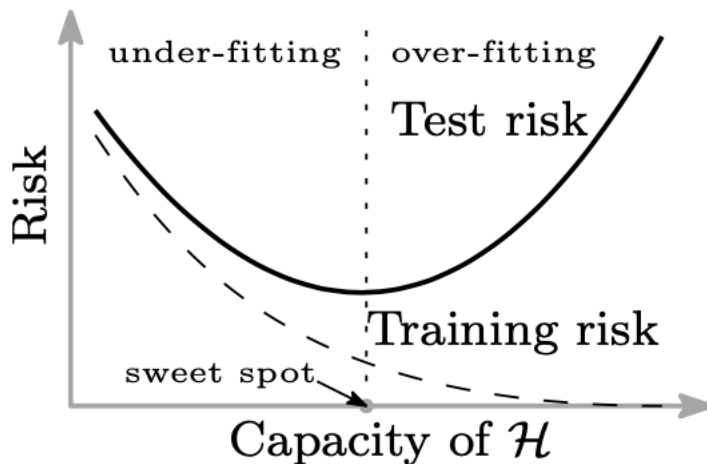
Call:
lm(formula = "y ~ .", data = X)

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
```

2.5 Overfitting and the Bias-Variance Tradeoff

Wiki

- Bias: Underfitting
- Variance: Overfitting - the model overfits peculiarities of the data sample



This is the perspective of classical statistics:

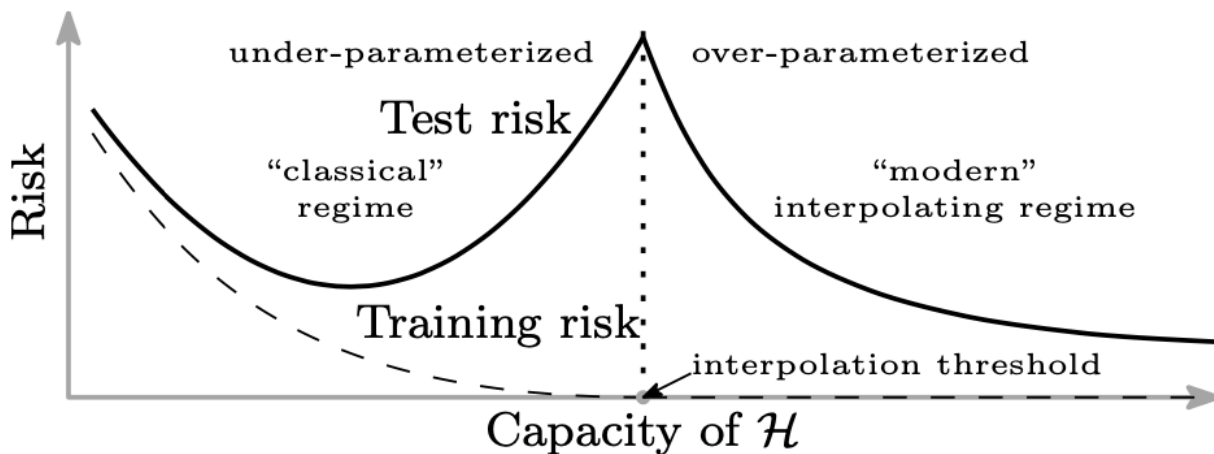
- more parameters lead to overfitting
- the results of models with many parameters are not reliable (due to the high variance)
- with more parameters it's harder for single parameters to reach the significance threshold (statistical testing)
- smaller models are better (epistemology: prefer simpler solutions if they are as good as the more complex ones)

But neural networks are heavily over-parameterized with far more weight-parameters than independent samples in the training data. How comes they generalize quite well?

a modern perspective on overfitting

Following [Belkin et al., 2019](#) and [Dar et al., 2021](#):

- When we have as many parameters as data samples, the number of solutions is very constrained. The model has to “stretch” to reach the interpolation threshold with a limited capacity. This explains the weird loops the polynomial makes.
- When we have more parameters than data points the space of interpolating solutions opens-up, actually allowing optimization to reach lower-norm interpolating solutions. These tend to generalize better, and that's why you get the second descent on test data.



For the interested reader:

- [On the Bias-Variance Tradeoff: Textbooks Need an Update](#)
- [Double Descent](#)
- [Deep Double Descent](#)
- [Are Deep Neural Networks Dramatically Overfitted?](#)

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, \dots 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.

3.1 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 larger for extrem values of b , 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

$$\sum_i^n [y_i - (a + b \cdot x_i)]^2 \tag{1}$$

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

sind wie Antagonisten. Der Koeffizient b darf nur gross werden, wenn er es vermag (1) stark zu verkleinern, so dass der Zugewinn in (1) den Strafterm in (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 Einsen 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 linear 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 einzelnen 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 square matrix\n is
↪\n
      + 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 square matrix\n is
↪\n
      + f'{np.linalg.matrix_rank(np.dot(X_7.T, X_7))}')

print(f'By adding a small amount (+2) to the diagonal of the matrix, it is of full_
↪rank\n again: '\n
      + 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('\n to 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 (+2) 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])
```

3.1.1 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 approach 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')

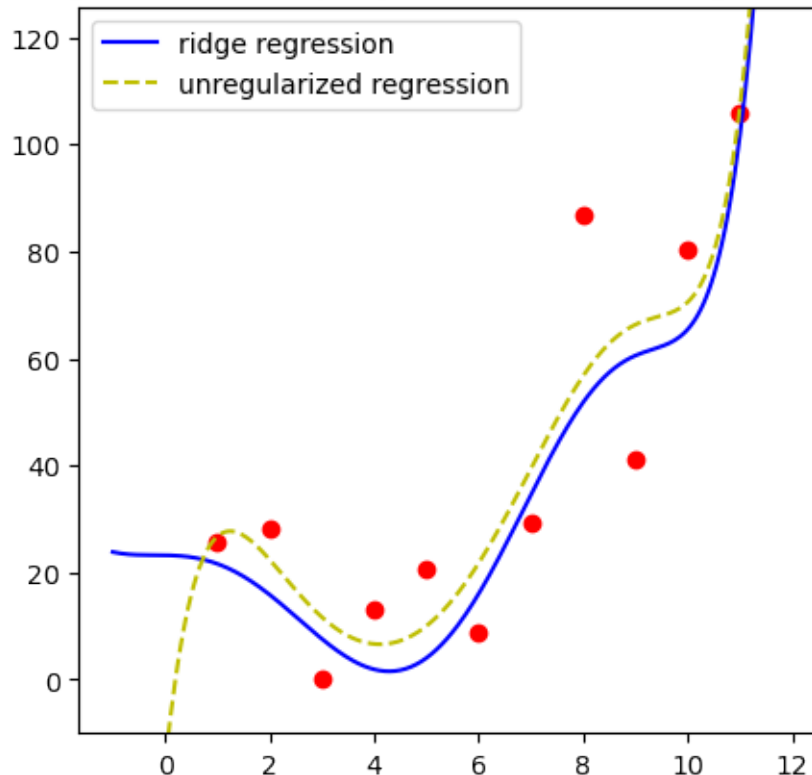
X = np.c_[X, X**2, X**3, X**4, X**5, X**6, X**7]
# here is the necessary standardization:
Xc = X - np.mean(X, axis=0)

# 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]]
```

ridge regression for polynome of 7th degree and $\lambda = 2$ 

Now, it becomes clear why Ridge Regression was invented before Lasso Regression. We have an analytical solution. Ridge is nearer to 'old school statistics' than Lasso is.

3.2 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 Gegensatz 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. Iterative Verfahren haben sich erst sehr spät durchgesetzt - nicht zuletzt wegen der Rechenleistung die sie benötigen.

kurzer Einschub: klassische Statistik vs. Machine Learning

- Mathematisch liess sich lange Zeit nur ein lineares Gleichungssystem zuverlässig lösen (Rechenpower). Deshalb wurde Ridge-Regression auch vor Lasso-Regression erfunden. Für ersteres Verfahren gibt es eine analytische Lösung.

- Das lineare Modell setzt voraus, dass alle Variablen darin voneinander unabhängig und normal verteilt sind. Dies trifft auf fast keinen Umstand in unserer Welt zu.
- Konfidenzintervalle und Signifikanzen sind das direkte Resultat dieser Annahmen und der damit verbundenen mathematischen Lösung - der Inversion der Kreuzprodukt-Matrix - so wie wir das besprochen haben.
- **“Overfitting”** ist der Begriff, der verwendet wurde, wenn das verwendete mathematische Verfahren die Daten der Stichprobe zu genau repräsentiert und auf neue Daten schlecht generalisiert.
- Leider wurde **“overfitting”** oft gleichbedeutend mit zu vielen Variablen verwendet.
- Wird die Grösse der Parameter (die Norm) klein gehalten (Ridge, Lasso) so tritt **“overfitting”** nicht auf.
- Mittlerweile gibt es zuverlässige Verfahren, die overfitting zu verhindern wissen. Da die Modellannahmen in den Wissenschaften oft nur getroffen wurden, weil es für diese eine analytische Lösung gibt, müssten eigentlich viele Lehrbücher umgeschrieben werden.
- Die Verfahren mit vielen Variablen finden in vielen Anwendungen sehr gute Lösungen und haben einige Anwendungsfelder geradezu revolutioniert (Sprach- und Bilderverarbeitung).
- Wissenschaftstheoretisch sind die neuen Verfahren nicht hinreichend, aber auch die alten Verfahren sind von geringem Nutzen, wenn die Annahmen falsch sind.
- Es wird Zeit, bisherige klassische Statistik und Verfahren des maschinellen Lernen miteinander zu versöhnen. Eine neuere, umfassende Theorie muss entwickelt werden.
- see also: [statistical modeling: the two cultures](#)

3.2.1 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 unconstrained 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]

```

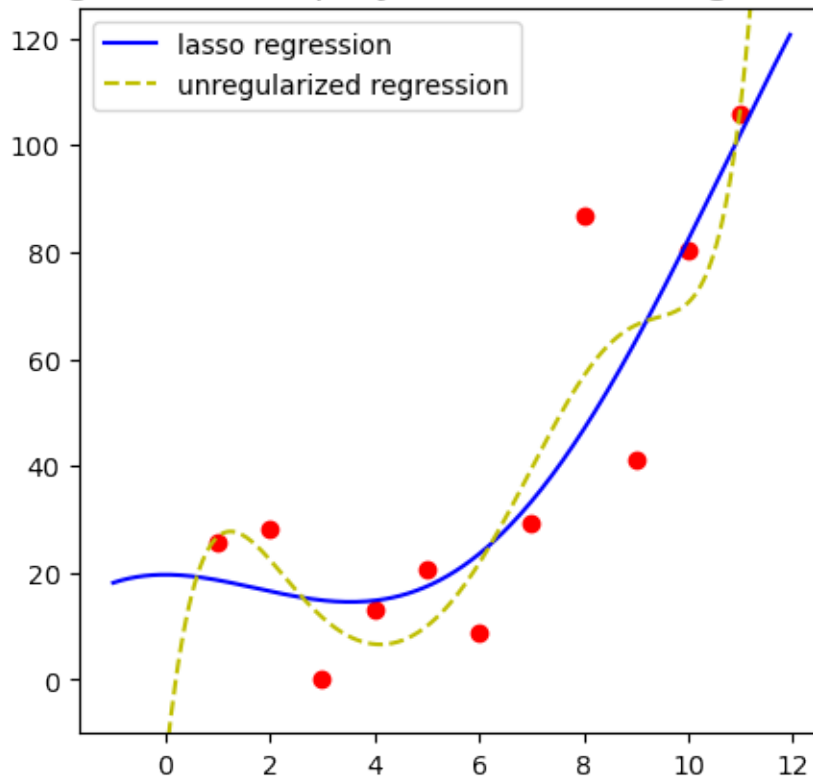
Ridge vs Lasso

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 $0.000747 = 7.47e - 04$.

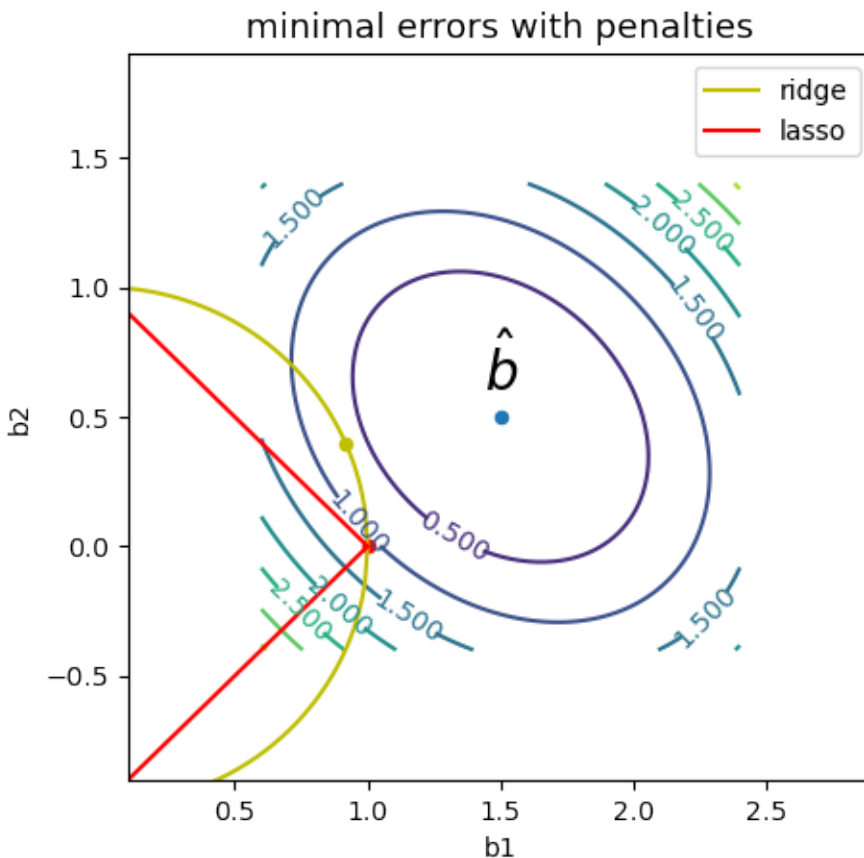
lasso regression for polynome of 7th degree and $\lambda = 2$



3.3 the difference between ridge and lasso

In der folgenden graphischen Darstellung haben die **wahren Koeffizienten** die Werte $b_1 = 1.5$, $b_2 = 0.5$. 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 Strafterme ($b_1^2 + b_2^2$ im Falle von Ridge und $b_1 + b_2$ im Falle von Lasso) den Wert von 1.0 nicht übersteigen. Die Lösung, die den **wahren Koeffizienten** am nächsten 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 ($b=1.5$, $b_2=0.5$) am Nächsten 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: 0.9191919191919192 and 0.3938098725175338
optimal coefficients of the lasso solution: 1.0 and 0.0
```

3.4 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 -Parameter, 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 parameter λ und α ist wie folgt:

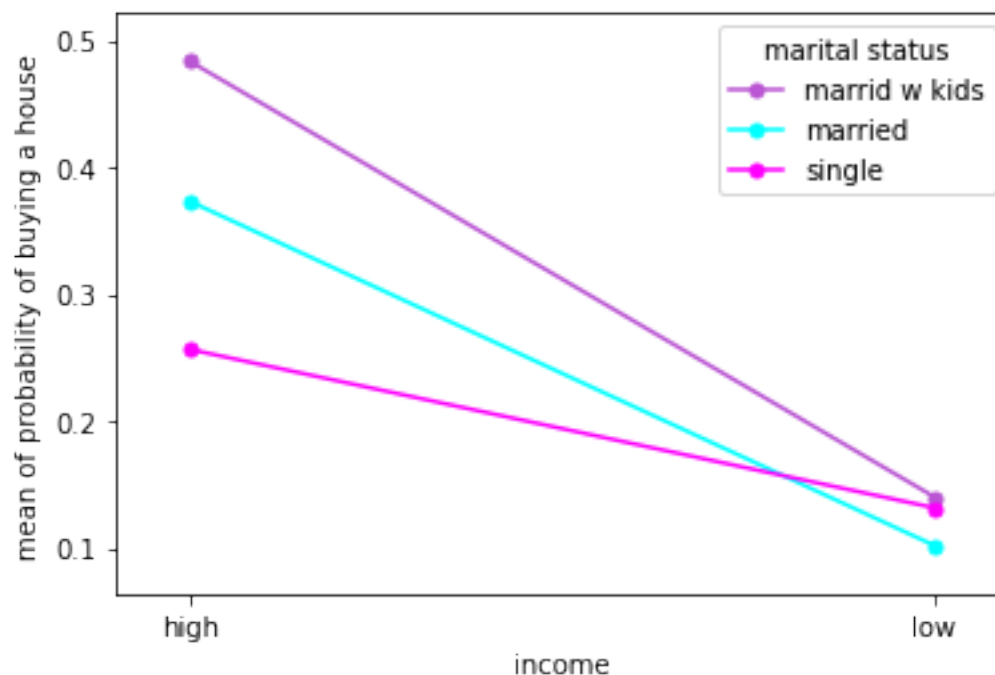
- λ bestimmt das generelle Mass an regularisation
- α gibt das Verhältnis an, mit dem diese beiden Strafterme in die regularisation einfließen sollen

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

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 unserem 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”: **Der Effekt der beiden Variablen zusammen ist mehr als die Summe der Effekte der einzelnen Variablen.**



Das obige Beispiel beinhaltet kategoriale 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 überhaupt in 2D dargestellt werden kann, muss 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:

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

`np.random.normal(loc=0, scale=4, size=n)` ist der Random-Error-Term, den wir hinzufügen, damit die Daten nicht alle auf einer Linie 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

Folgende haben wir also die Koeffizienten:

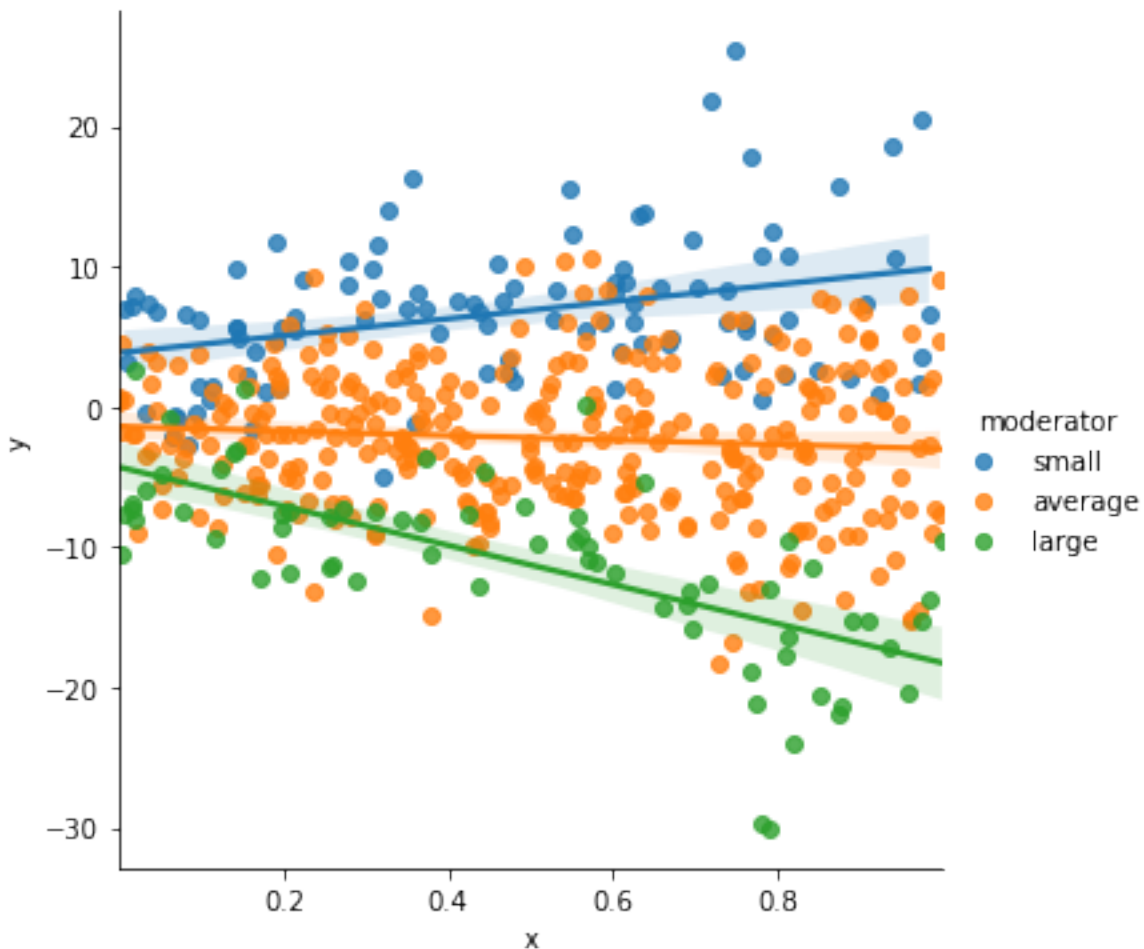
- $b_x = 2$
- $b_m = -2$
- $b_{x \cdot m} = -7$

```
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 Hinzunahme 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\n
↪values \'
    + f'we used for generating the data')
```

```
without considering the interaction, the mse is: 18.528824692075656
considering the interaction, the mse drops to: 14.89943728747419

the coefficients are given by [ 1.15051725 -2.66712516 -5.93210293]; compare these
↪values
to the values we used for generating the data
```

4.1 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 Beobachtungen 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 aufnehmen: $\binom{210}{2} = 21945$
- 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 Beispiel des Boston 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 Ordnung, 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.**

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 unser Modell aufnehmen. Wie können wir nun die Güte unserer Schätzung beurteilen? Hier sind mindestens zwei verschiedene Fragen denkbar:

- Wie sicher sind wir mit Hinblick auf die geschätzten Koeffizienten \mathbf{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:

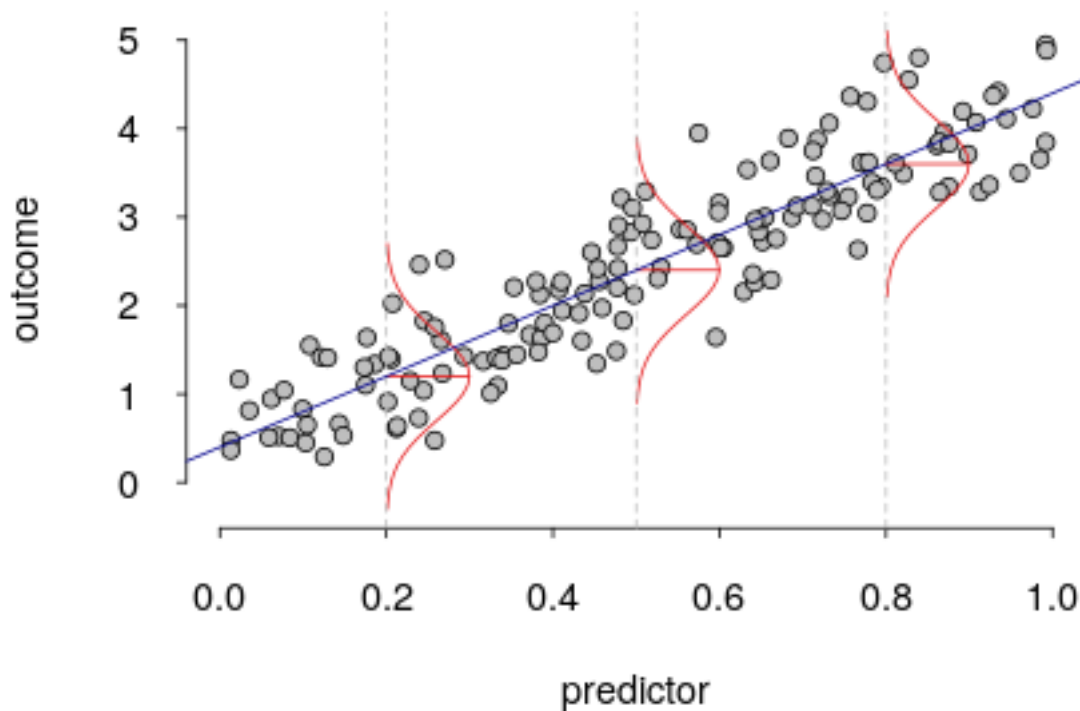
- Wie sehr ist die ‘mean response’, die Vorhersage unsere Regressions-Funktion von der Stichprobe abhängig. Variiert Erstere sehr stark und umfasst unter Umständen sogar den Wert 0, dann können diese Effekte (Koeffizienten) nicht interpretiert werden.
- Wie sehr können Beobachtungen y für eine gegebene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

5.1 Recap of assumptions underlying regression

Dies sind Linearität (der Zusammenhang einer Variablen und der abhängigen Variablen ist linear, d.h. der selbe Steigungsparameter 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. **Vorhersage Intervall (prediction interval):** Dies ist das Intervall, in welchem mit $(1-\alpha)\%$ Wahrscheinlichkeit die beobachteten y Werte zu unseren vorhergesagten Werten \hat{y} liegen. Dieses Intervall ist symmetrisch um die Regressionsfunktion - was natürlich aus den Voraussetzungen der linearen Regression folgt. Der Standardfehler der Vorhersage ist gegeben durch:

$$\hat{\sigma}_e = \sqrt{\frac{1}{N - (p + 1)} \sum_i^N e_i^2},$$

hier ist p die Anzahl der Parameter im Modell (der zusätzliche Parameter $+1$ kommt vom intercept); e_i sind die Vorhersage-Fehler, die Residuen, also die Differenz aus unseren vorhergesagten \hat{y}_i und den beobachteten Werten y_i . Das Konfidenz-Intervall ergibt sich zu:

$$CI_i = y_i \pm t_{1-\alpha/2, N-p} \cdot \hat{\sigma}_e.$$

Hier ist $t_{1-\alpha/2, N-p}$ der Wert der Student-t-Verteilung für das Konfidenzlevel von $1 - \alpha/2$ und $N - p$ Freiheitsgraden. Der Wert von α gibt an, wie sehr wir uns mit dem Konfidenzintervall gegen falsche Entscheidungen absichern wollen. Wollen wir beispielsweise mit 95% Sicherheit den Bereich angeben können, in dem die beobachteten Werte liegen, dann müssen wir als untere Konfidenzgrenze den Wert bestimmen unterhalb dessen nur mit einer Wahrscheinlichkeit von 2.5% die beobachteten Werte liegen und als obere Konfidenzgrenze den Wert unterhalb dessen mit einer Wahrscheinlichkeit von 97.5% die beobachteten Werte liegen. So machen wir nur in 5% aller Fälle einen Fehler, $\alpha = 0.05$ und weil das Konfidenzintervall symmetrisch ist benötigen wir den Wert $1 - \alpha/2$ damit wir von beiden Enden 2.5% abschneiden.

1. **Mean Prediction Confidence interval:** In ähnlicher Weise können wir ein Konfidenzintervall für unsere durchschnittliche Vorhersage $\hat{\bar{y}}$ bestimmen. Wir erinnern uns, dass die Regressions-Funktion unsere Vorhersage ist und die Daten um diese normal verteilt sein sollten. Weil unsere Stichprobe aber nur eine Momentaufnahme eines Ausschnitts aller möglichen Werte ist, die wir erheben könnten, wird die Regressions-Funktion je nach Stichprobe variieren. Das Konfidenz-Intervall gibt an, in welchem Bereich die Regressions-Funktion mit grosser Wahrscheinlichkeit liegen würde, könnten wir alle Daten erfassen (die gesamte Population). Das Konfidenzintervall ist nicht für alle Werte von x gleich weit. Dort wo wenige Messwerte vorliegen kann der genaue Verlauf schlechter geschätzt

werden als dort wo wir eine breitere Datenbasis für die Schätzung haben. Nahe dem Mittelwert von x , also bei \bar{x} sollte unsere Schätzung immer genauer sein als nahe den Extremwerten. Natürlich gehen wir wieder von normalverteilten x Werten aus.

1. **CI for regression coefficients:** Auch diese Intervall ist schwierig zu bestimmen. Es gibt die obere und untere Grenze für unsere Regressions-Koeffizienten \mathbf{b} . Die Interpretation dieser Koeffizienten findet vor allem in der Wissenschaft statt. Es kann getestet werden, ob ein a priori postulierter Effekt tatsächlich vorliegt oder nicht. Umfasst das Konfidenzintervall für einen Koeffizienten b den Wert Null, so kann nicht ausgeschlossen werden, dass der Effekt in der Stichprobe nur rein zufällig zu Stande kommt. Beispielsweise könnte folgende Fragestellung hiermit untersucht werden: "Hat die Schliessung von Schulen und Universitäten einen signifikanten Einfluss auf die Reproduktions-Zahl R_0 oder nicht". Dies ist typischerweise nicht die Art von Fragestellung, mit der sich Data Scientists beschäftigen.

Im nachfolgenden Beispiel sehen wir die typische Ausgabe eines klassischen, statistischen Ansatzes. In der Mitte sehen wir die Konfidenz-Intervalle für die Regressions-Koeffizienten, `const` (intercept) und `x1`, d.h. der Koeffizient der Variablen x_1 , also b_1 . Der intercept ist nicht signifikant, weil das Konfidenzintervall $([-43.351, 26.370])$ den Wert 0 umfasst. Der Koeffizient b_1 für die Variable x_1 ist aber signifikant von 0 verschieden. Sein Konfidenzintervall ist $[2.939, 13.219]$.

Bei grossem Interesse für klassische statistische Modelle kann ich für python diese [Quelle](#) empfehlen.

```
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:	Thu, 16 Feb 2023	Prob (F-statistic):		0.00616		
Time:	21:27:22	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

(continues on next page)

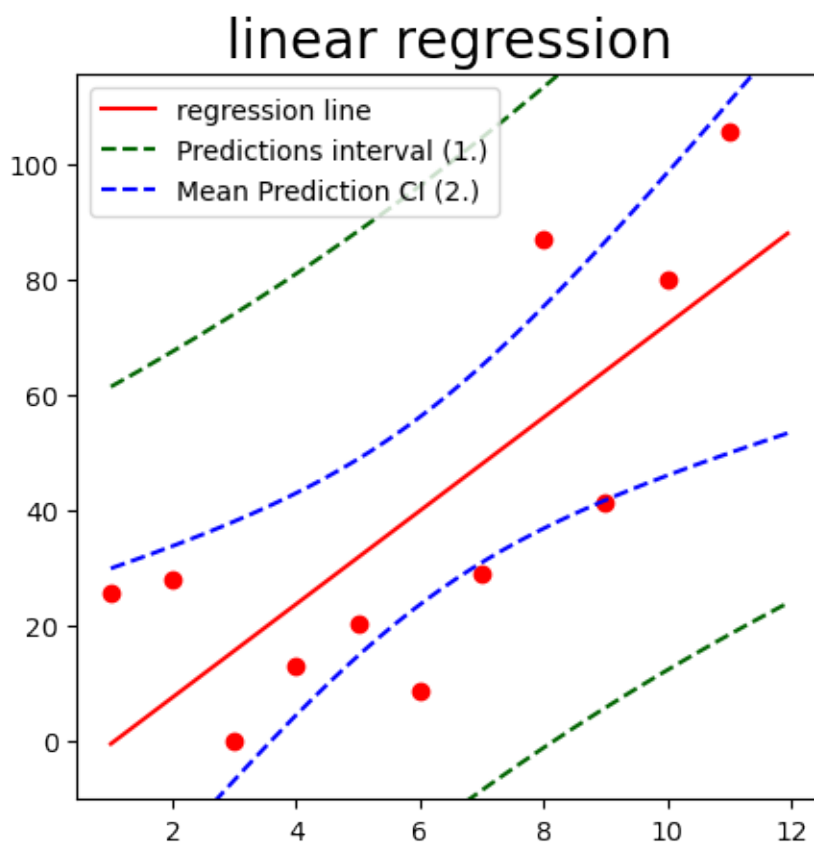
(continued from previous page)

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)



Als nächstes berechnen wir die Konfidenzintervalle für die Regression mit einem quadratischen Term. Hierbei fällt auf, dass zwar jetzt der quadratische Term x_2 signifikant ist (Intervall $[0.247, 3.125]$), nicht mehr aber der x_1 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()
```

(continues on next page)

(continued from previous page)

```

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']]

```

```

=====
                        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:                          Thu, 16 Feb 2023    Prob (F-statistic):      0.00224
Time:                          21:27:29    Log-Likelihood:          -45.820
No. Observations:              11    AIC:                     97.64
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

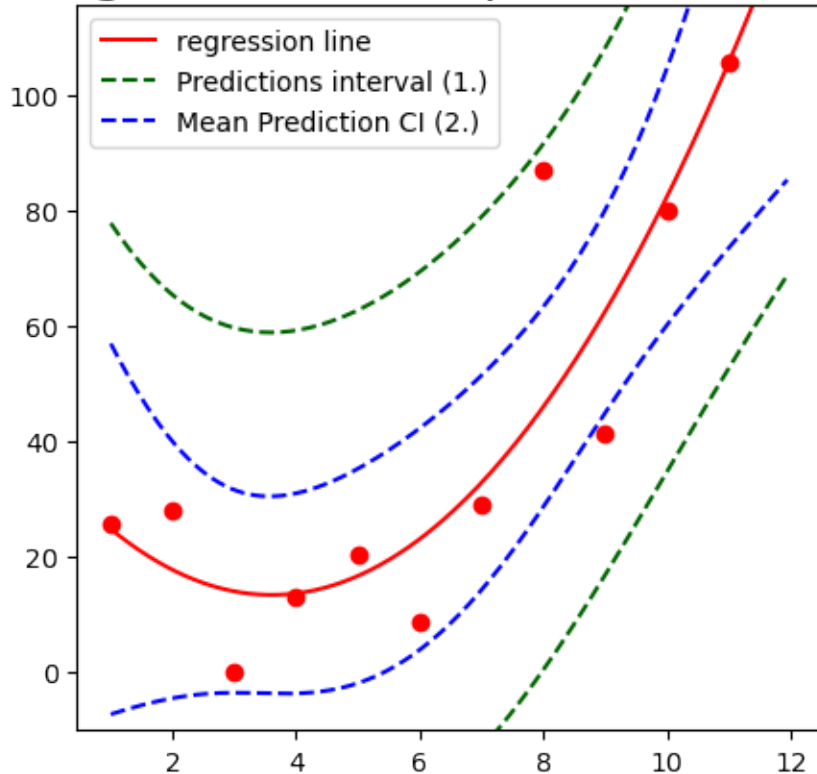
```

=====
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)
```

regression with quadratic term



5.2 Bootstrap

Die Daten, mit denen ein Data Scientist normalerweise arbeitet, erfüllen meist nie die Voraussetzungen der Linearen Regression. Deshalb können wir auch die Theory zu den Konfidenzintervallen nicht anwenden - schliesslich beruht sie auf den Annahmen wie normalverteilte Daten. Eine robuste, parameter-freie Alternative ist der **Bootstrap**. Gewissernahmen ziehen wir uns an den eigenen Haaren aus dem Schlamassel:

- Wir betrachten unsere Stichprobe als die Gesamtheit (Population) der Daten.
- Nun ziehen wir wiederholt und mit Zurücklegen neue Stichproben aus dieser Stichprobe.
- Für jede dieser Stichproben wird das Modell angepasst und die relevanten Statistiken werden gespeichert.
- Abschliessend finden wir in unseren gespeicherten Statistiken das 2.5% Quantil (der Wert, unter dem nur 2.5% der Beobachtungen liegen) und das 97.5% Quantil (der Wert über dem nur noch 2.5% der Beobachtungen liegen). Diese Werte teilen wir als untere und obere Grenze des Konfidenz-Intervalls mit, bei einem Konfidenz-Level von $\alpha = 0.05$.

Nachfolgendes Code-Beispiel veranschaulicht den Vorgang:

- `sampler = (choices(indices, k = len(indices)) for i in range(200))` erzeugt einen Generator, der 200 Mal eine Zufallsstichprobe zieht.
- `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)` iteriert über den Generator und passt insgesamt 200 mal das Modell an und macht eine Vorhersage für kontinuierliche x-Werte im Bereich von 1 bis 12. Diese Vorhersagen werden in einem numpy-array (`np.array`)

gespeichert und zu Schluss die Funktion `np.percentile` auf die 200 Vorhersagen angewendet. Somit erhalten wir für den x-Bereich von 1 bis 12 die Intervall-Grenzen für die mean-prediction, d.h. die Regressions-Funktion

```
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])

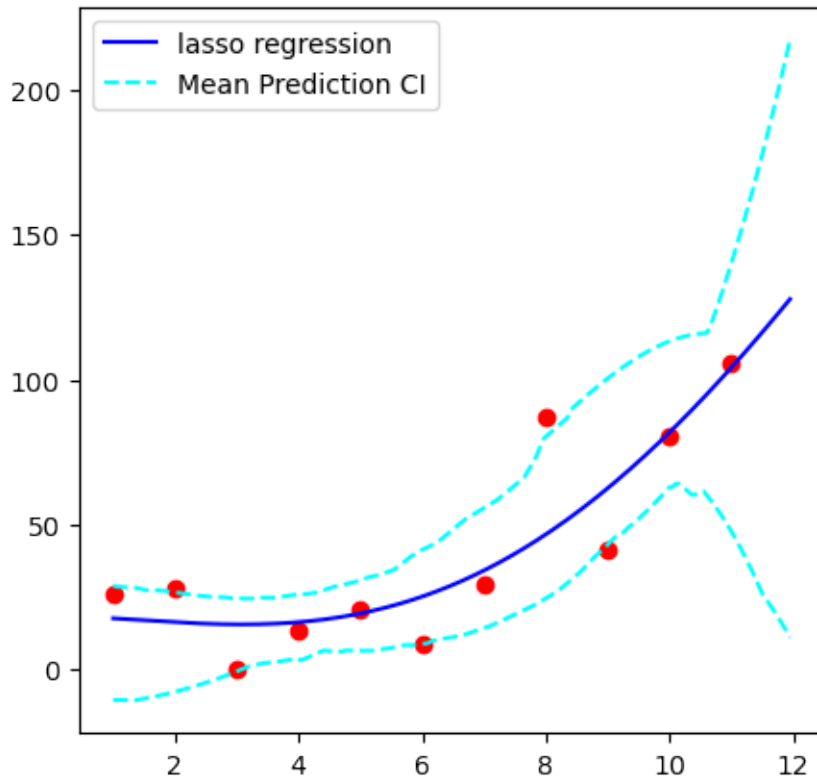
sampler = (choices(indices, k = len(indices)) for i in range(200))

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 0x7ffa9764ef60>
```

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



EXTENSION: LOGISTIC REGRESSION AND THE GLM

Es gibt andere Modelle, die eng verwandt mit der hier besprochenen Linearen Regression sind. Das Prominenteste unter ihnen ist die **Logistische Regression**. Diese Modell gehört zu dem “**Verallgemeinerten Linearen Modell**” (im engl. **generalized linear model** (GLM)). Diese Modelle dürfen nicht mit dem “**Allgemeinen Linearen Modell**” (im engl. **general linear model**) verwechselt werden. Letzteres parametrisiert eine Varianzanalyse als ein lineares Modell mit Dummy-Variablen. Das Verallgemeinerte Lineare Modell erweitert die Lineare Regression um Modelle, deren Fehler nicht normalverteilt sind. [Dieser Artikel](#) in der Wikipedia gibt weitere Auskunft.

6.1 exponential family of distributions

Aus der Perspektive der Modernen Statistik beinhaltet das Verallgemeinerte Lineare Modell verschiedene Lineare Modelle, unter anderem das der klassischen linearen Regression. Eine Verteilung, die in der “exponential family” von Verteilungen ist, kann immer folgendermassen geschrieben werden:

$$f(y|\theta) = \exp \left(\frac{y\theta + b(\theta)}{\Phi} + c(y, \Phi) \right), \quad (1)$$

wobei θ als Kanonischer Parameter bezeichnet wird, welcher eine Funktion von μ ist dem Mittel. Diese Funktion wird als Kanonische Link-Funktion bezeichnet. Wie wir später an einem Beispiel sehen werden, ist es genau diese Funktion welche die Beziehung zwischen der abhängigen Variablen und den unabhängigen Variablen linearisiert. Der Vollständigkeit halber: $b(\theta)$ ist eine Funktion des Kanonischen Parameters und ist somit ebenfalls von μ abhängig. Φ wird als Streuungsparameter bezeichnet und $c(y, \Phi)$ ist eine Funktion, die sowohl von beobachteten Daten wie auch dem Streuungsparameter abhängig ist.

6.1.1 Normalverteilung

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

$\mu = \theta(\mu)$, d.h. μ ist der Kanonische Parameter und die Link-Funktion ist die Identitäts-Funktion. Der Mittelwert kann also ohne weitere Transformation direkt modelliert werden, so wie wir es in der klassischen Linearen Regression machen. Der Streuungsparameter Φ ist durch σ^2 , die Varianz gegeben. Dies ist die klassische Lineare Regression normalverteilter Variablen

6.1.2 Poisson distribution

Die Poisson-Verteilung gehört ebenfalls der exponential family von Verteilungen an:

$$\begin{aligned} f(y|\mu) &= \frac{\mu^y e^{-\mu}}{y!} = \mu^y e^{-\mu} \frac{1}{y!} \\ &= \exp(y \log(\mu) - \mu - \log(y!)) \end{aligned}$$

Die Link-Funktion ist hier $\log(\mu)$. Beachte bitte, dass die Poisson-Verteilung keinen Streuungsparameter besitzt.

6.1.3 Bernoulli distribution \Rightarrow logistic regression

Zuguter Letzte, die Bernoulli Verteilung, von der wir die Logistische Regression ableiten können. Die Bernoulli Verteilung eignet sich um binäre Ereignisse zu modellieren, die sich gegenseitig ausschliessen. Ein klassisches Beispiel ist der wiederholte Münzwurf. Die Wahrscheinlichkeit für ‘Kopf’ wird mit π bezeichnet, die für ‘Zahl’ mit $(1 - \pi)$. Hiermit lässt sich die Wahrscheinlichkeit berechnen, mit einer fairen Münze bei 10 Würfeln eine bestimmte Sequenz mit genau 7 Mal ‘Kopf’ zu erhalten:

$$\pi^7 (1 - \pi)^3 = 0.5^7 0.5^3 = 0.5^{10} = 0.0009765625 \quad (2)$$

Vorsicht, wenn wir die Wahrscheinlichkeit für Sequenzen mit genau 7 Mal Kopf berechnen wollen, benötigen wir noch den Binomial-Koeffizienten, der uns die Anzahl an möglichen Sequenzen mit 7 Mal ‘Kopf’ angibt.

Jetzt zeige ich, wie wir die Bernoulli Verteilung so umschreiben können, dass man ihre Zugehörigkeit zur exponential family von Verteilungen erkennt:

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

sich die Link-Funktion zu $\log(\frac{\pi}{1-\pi})$ ergibt. Diese Funktion wird auch als Logit-Funktion bezeichnet. Die Umkehrfunktion der Logit-Funktion ist die **Logistische Funktion**. Es ist also die Logit-Funktion, die als lineare Kombination der unabhängigen Variablen modelliert wird. $\log(\frac{\pi}{1-\pi}) = a + b_1 x_1 + \dots + b_j x_j$. Wenn wir den rechten Teil dieser Gleichung in die Logistische Funktion einsetzen erhalten wir die geschätzten Wahrscheinlichkeiten:

$$P(y = 1|x) = \frac{\exp(a + b_1 x_1 + \dots + b_j x_j)}{1 + \exp(a + b_1 x_1 + \dots + b_j x_j)} \quad (3)$$

Somit haben wir also gezeigt, dass das klassische Lineare Regressions-Modell nur ein Spezialfall einer grossen Anzahl von Modellen ist, deren Verteilungen alle in der exponential family enthalten sind. (Für eine vollständige Abhandlung dieses Themas: https://en.wikipedia.org/wiki/Generalized_linear_model.)

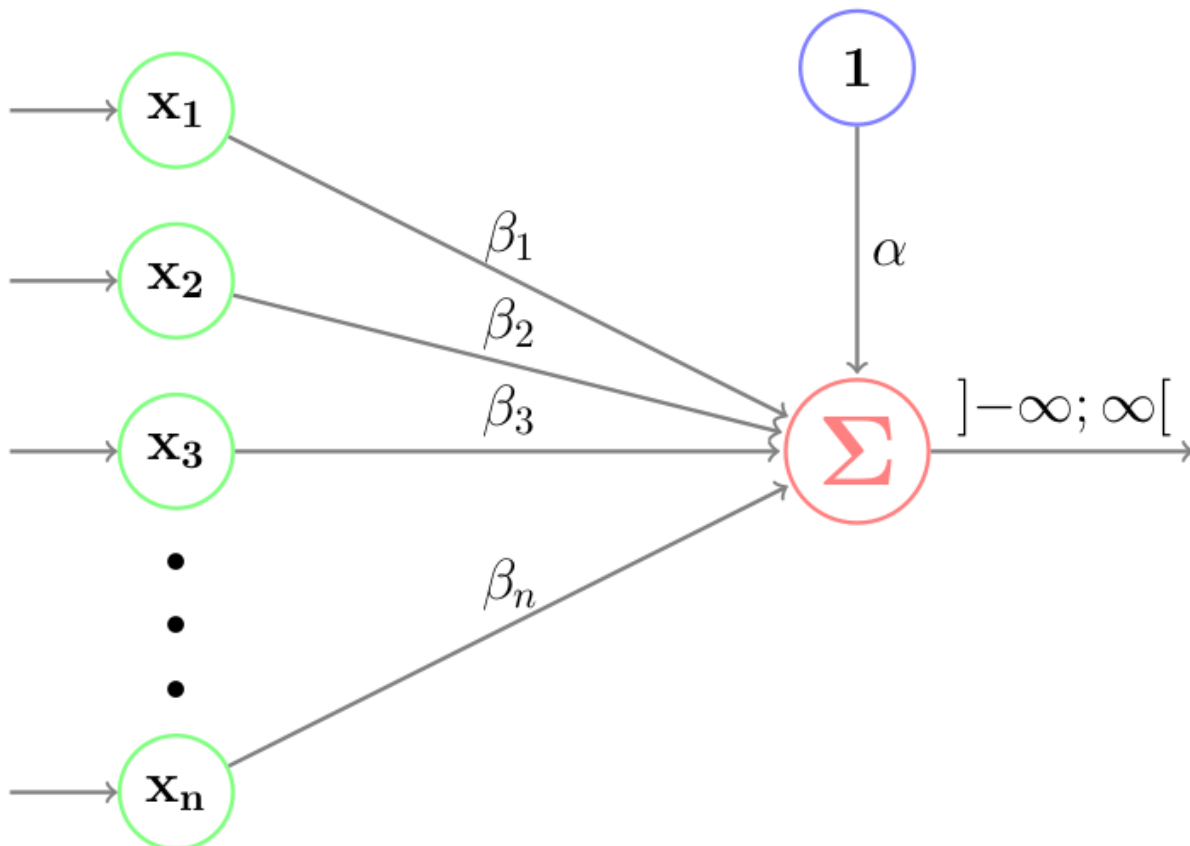
6.1.4 GLMNET

In der statistischen Programmiersprache R gibt es eine library die ‘glmnet’ genannt ist. Dieses Paket implementiert das ElasticNet für das Verallgemeinerte Lineare Modell und nicht nur für die klassische Lineare Regression. https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html

Es gibt auch ein python package welches den exakt gleichen Fortran-Code verwendet: **glmnet-python**. Es gibt ein paar kleine Unterschiede zu der Version von ElasticNet wie sie in `scikit-learn` implementiert ist <https://pypi.org/project/glmnet-python/>

NEURAL NETWORK

Es ist auch möglich Neuronale Netzwerke unter dem Blickwinkel der Linearen Regression zu betrachten. Ein Netzwerk mit nur einer Eingabe-Schicht und einem Neuron wird als Perceptron bezeichnet. Die Aktivierungs-Funktion dieses Neurons ist entweder die Identitäts-Funktion, so wie in der klassischen Linearen Regression oder die Logistische Funktion wie in der Logistischen Regression. In letzterem Fall soll das Perceptron Wahrscheinlichkeiten für binäre Ereignisse bestimmen.



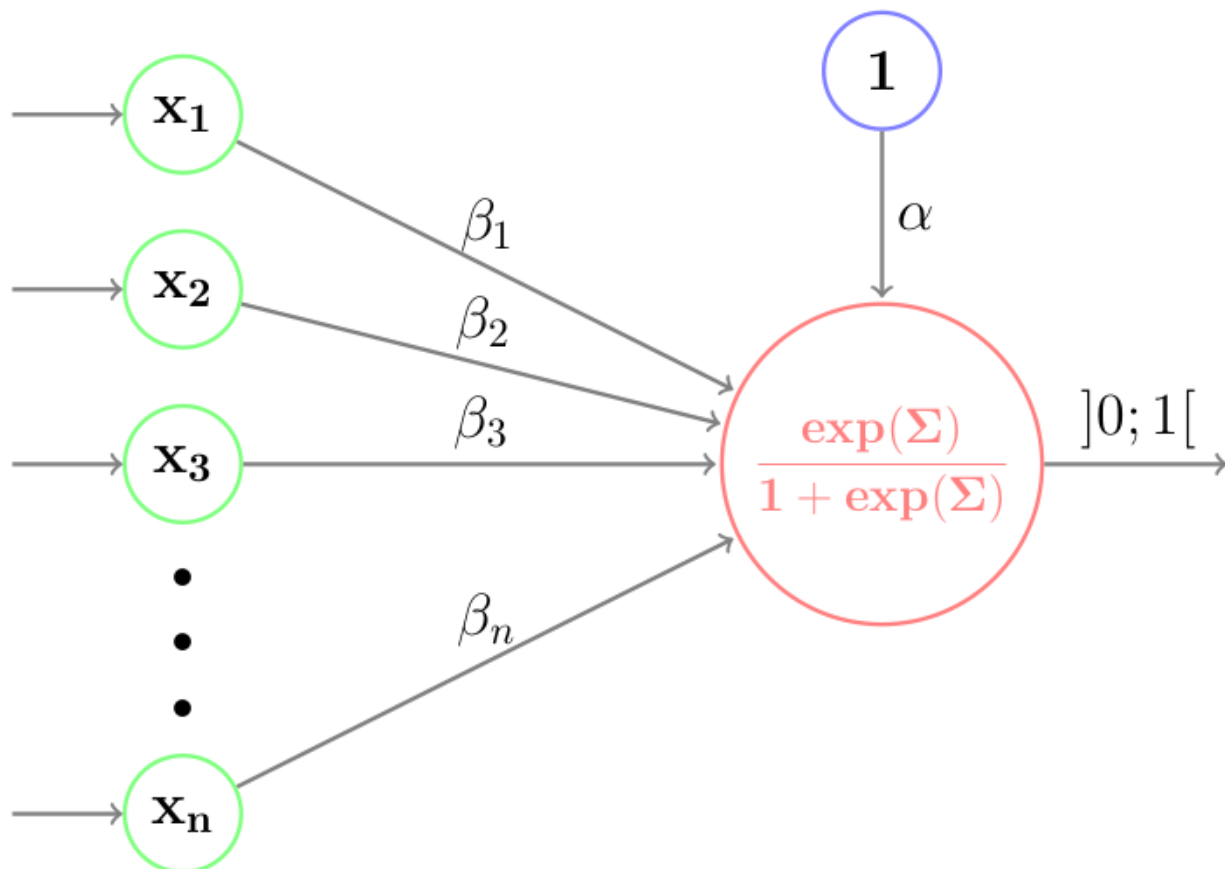
7.1 classical linear regression

Im Jargon der neural network community werden unsere b -Koeffizienten als **Gewichte** bezeichnet. Der intercept α heisst **bias**. Erinnert Euch, dass wir den intercept α in den Vektor β der b -Koeffizienten aufgenommen haben, indem wir eine Einser-Spalte in die Variablen-Matrix \mathbf{X} eingefügt haben. Wir könnten also Schreiben:

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

In der obigen Graphik könnt ihr sehen, dass im Perceptron die Input-Variablen mit den Gewichten der Verbindungen multipliziert werden und dass der konstante Wert α hinzu addiert wird. Wie in der Linearen Regression werden diese Produkte dann aufsummiert.

Im Kontext Neuronaler Netzwerke wird der Vektor β als Netzwerk-Gewichte bezeichnet und wird mit \mathbf{W} angegeben. Wir hatten gelernt, dass Vektoren mit kleinen Buchstaben bezeichnet werden. In einem richtigen Neuronalen Netz haben wir in einer Schicht viel Perceptrons nebeneinander. Alle erhalten aber den Input aus der darunter liegenden Schicht. Fügt man die Gewichts-Vektoren der einzelnen Neurone in eine Matrix zusammen, erhält man \mathbf{W} . Neuronale Netzwerke sind also eigentlich nur viele parallele und hintereinander geschaltete Regressionen, die sehr effizient mit Matrizen-Multiplikation gerechnet werden können.



7.2 logistic regression

Für die logistische Aktivierungs-Funktion schreiben wir:

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

Diese Funktion nähert sich asymptotisch der 0 für sehr kleinen Werte und der 1 für sehr grosse Werte.

7.2.1 Weight decay

In der Literatur zu Neuronalen Netzwerken wird der l_2 Strafterm als “weight decay” bezeichnet. Dieser Strafterm ist Teil des optimizers und nicht der einzelnen Neurone. Wie auch für Ridge Regression wird weight in die Fehler-Funktion mit aufgenommen:

$$L' = L + \lambda \sum_i w_i^2,$$

mit L als Loss (oder Fehler) und den w_i als die Gewichte der eingehenden Verbindungen der Neurone.

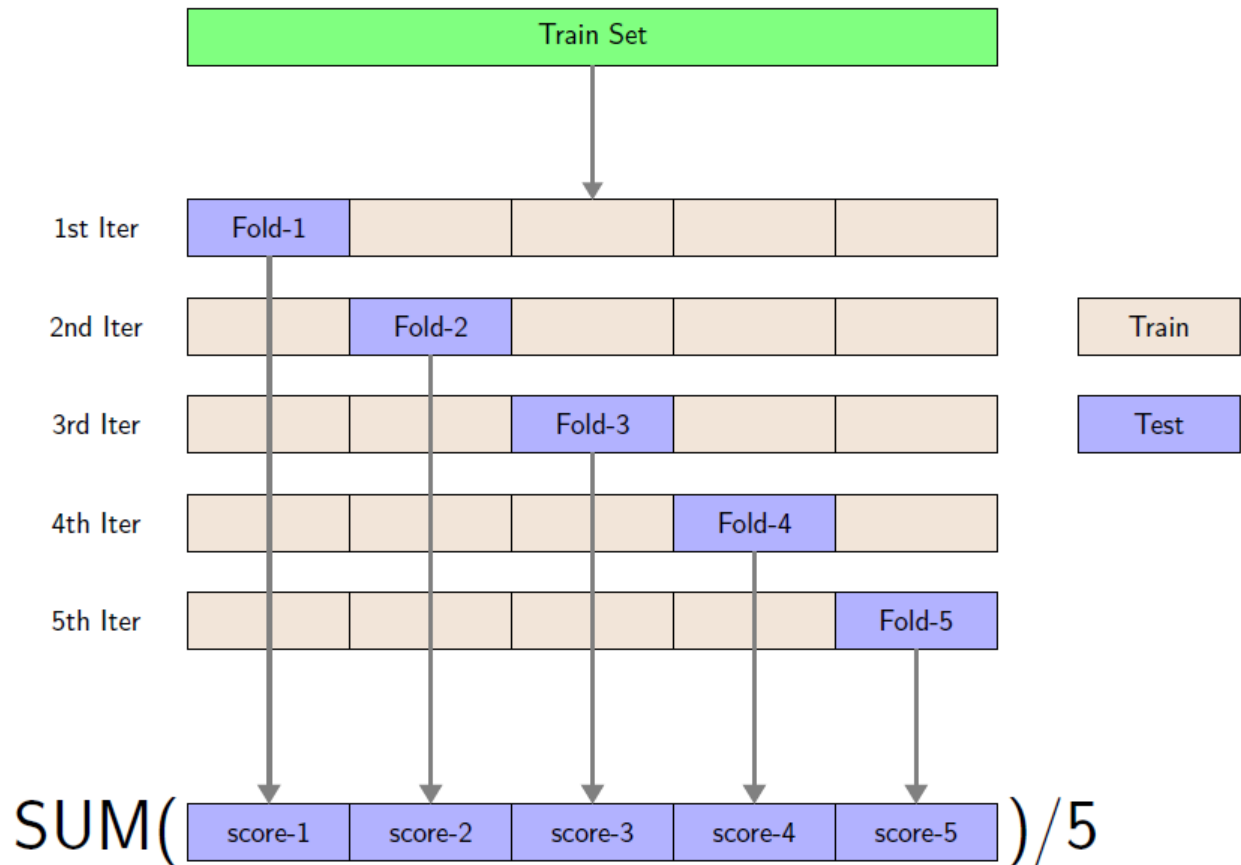
Part III

Cross Validation, Stacking and Mean Encoding

CROSS-VALIDATION

In most real-world applications we do not know the data universe, i.e. we do not know all possible data points that might be there. Our training data is possibly just a biased subsample of the population. When we fit our algorithm to such a subsample its performance will degrade, when applied to new, unseen data points. In order to have an idea, how well our algorithm will perform in such cases, we can use a cross-validation scheme: In the example below, a 5-fold cross-validation is illustrated.

- split the training data in 5 equal sized parts. In *sklearn* you can choose *StratifiedKFold*, that essentially tries to keep the percentages of all classes stable within each fold.
- train your algorithm on 4 folds and classify data in the 5th hold-out fold. Keep the performance on this fold.
- repeat the last step 4 more times and use each time another fold as your hold-out fold.
- at the end, you have 5 independent estimates of your algorithm's performance
- compute the mean of these 5 estimates for an overall estimate



8.1 example in python:

this is pseudo-code...algorithm and data are not defined here

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_auc_score

five_fold = StratifiedKFold(y=y_train, n_folds=5, shuffle=True)

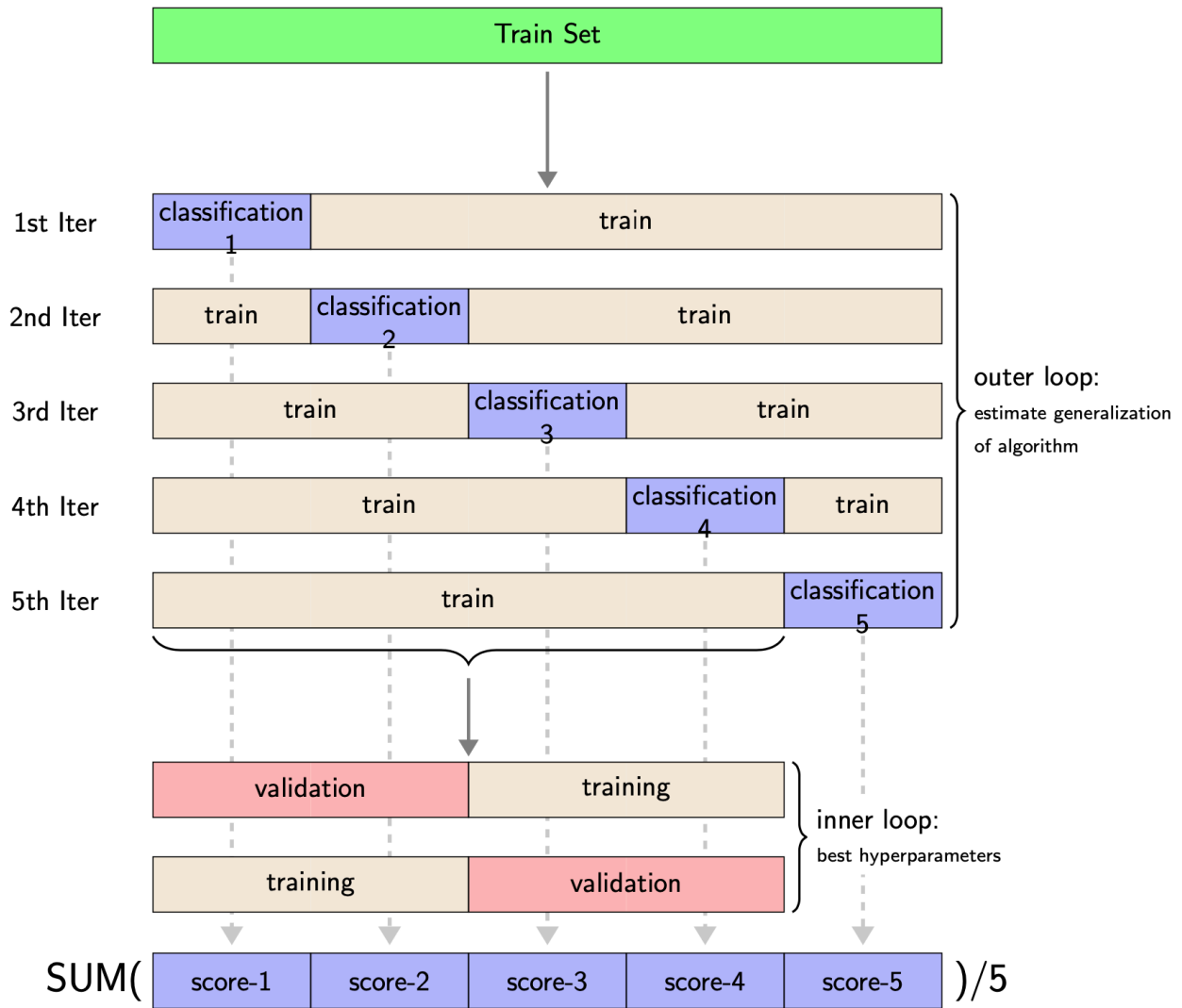
auc_scores = []
for train_idx, val_idx in five_fold:
    algorithm.fit(X_train[train_idx], y_train[train_idx])
    prediction = algorithm.predict(X_train[val_idx])
    auc_scores.append(roc_auc_score(y_true = y_train[val_idx], y_score = prediction))

print(f'mean area under the curve: {np.mean(auc_scores)}')
```


NESTED CROSS-VALIDATION

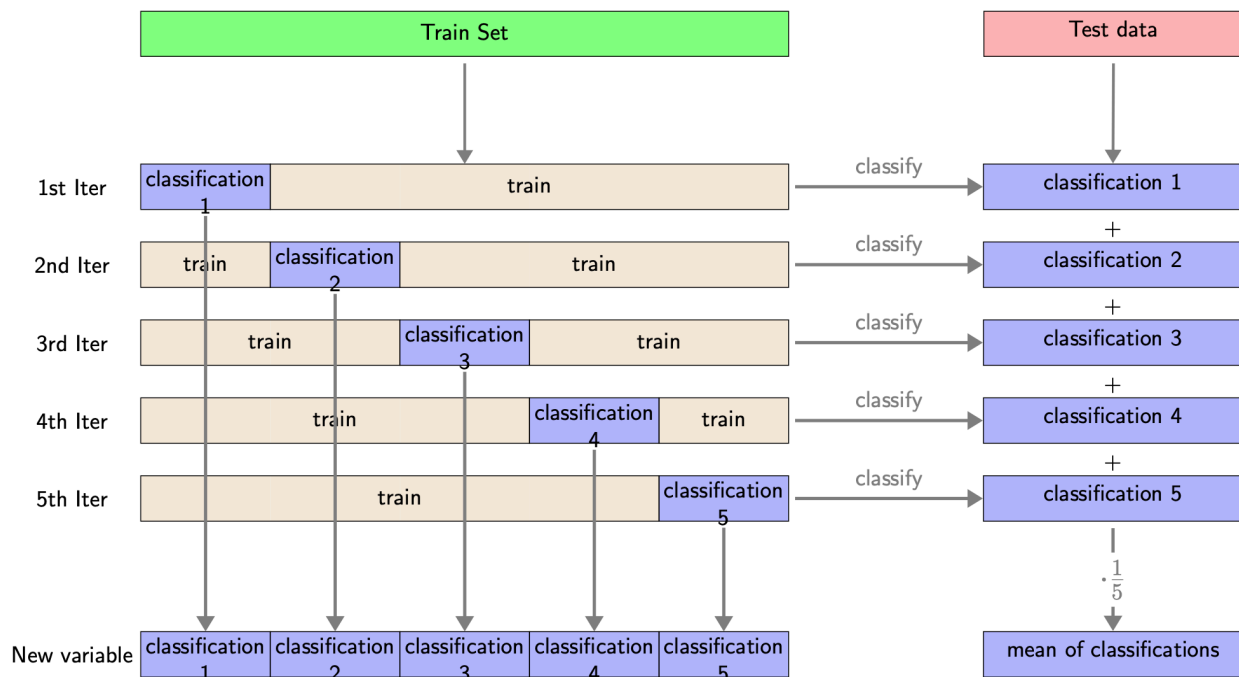
Why you should do it: [see a recent example for overfitting in hyper-parameter optimization \(HPO\)](#)

We learned how to estimate the true performance of an algorithm. Now, suppose we have to find the best parameters for an algorithm. We use a grid search and use for each parameter setting the cross-validation scheme as described. We then use the best parameter combination and report the estimate of the true performance on unseen data. Here, we introduce a bias, because we do an extensive search over different settings and in the end get the parameters with best cross-validation result on the training data. In other words, we probably fitted to peculiarities of the training data. The right way to do it is to use the trainings-folds as well for the parameter search. We then evaluate on the hold-out fold only once. This is called nested cross-validation because the cross-validation scheme for finding the best parameters is nested within our cross-validation that aims to get a unbiased estimate of the algorithm's performance:

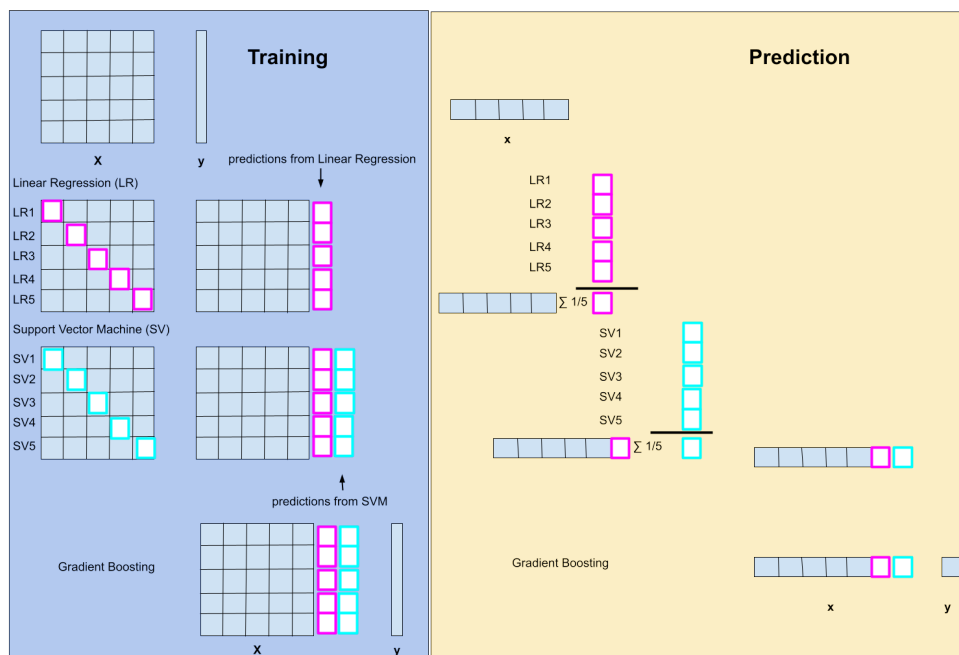


STACKING

Stacking is a technique that is related to cross-validation methods. It became very famous for kaggle competitions. Now, it is exhaustively used in every competition. Imagine, you intend to use the output of one algorithm as an input to just another algorithm. If you fit the first algorithm to all of the training data, the algorithm will certainly overfit. The generalization to unseen test data or to productive data will be bad. But, as we learned, we can mitigate these problems by the use of a cross-validation scheme. Stacking uses cross-validation for scoring the trainings data and algorithm ensembling for the test data. The important thing here to remember is, that the scores in the training data are **independent**. **The part of the training data scored in each iteration is not part of the data used to fit the algorithm.**



In the image below we see a careful plan; we add the predictions from a linear regression model and from a support vector machine to our variables. Here we always train on the training folds and predict the hold-out fold. In the example this results in five regression models and five svm models. Finally, a gradient boosting model is trained to make the actual prediction. This model now hopefully benefits from the predictions of the other models, which may have different strengths and weaknesses. In production, the predictions of the five models are then averaged and added to the actual data set. The final prediction is made by the gradient boosting model.



Please note that there are also less careful stacking schemes. In this case, the individual models are trained to the entire data set, which of course results in overfitting. To avoid this, the models are often trained only very briefly, in the hope of not getting into the overfitting regime.

```
X # TRAIN SET
y #

# models are needed for predicting future, unseen data
list_of_models = []
# this will be the new variable for our train-data-set
stacked_variable = np.zeros((len(y)), dtype=float)
# split data
five_fold = StratifiedKFold(y, n_folds=5, shuffle=True)

for train_idx, val_idx in five_fold:
    X_train, X_val = X.loc[train_idx, :], X.loc[val_idx, :]
    y_train, y_val = y[train_idx], y[val_idx]
    model.fit(X_train, y_train)

    # fill predictions into empty new variable
    stacked_variable[val_idx] = model.predict(X_val)

    list_of_models.append(model)

# add new, stacked variable
X = pd.concat([X, stacked_variable], axis=1)

# now for new, unseen data TEST_SET
XX # TEST SET
new_variable = np.zeros((XX.shape[0]))
for model in list_of_models:
    new_variable += model.predict(XX)/len(list_of_models)

XX = pd.concat([XX, new_variable], axis=1)
```

MEAN-ENCODING OR TARGET ENCODING:

11.1 only for classification

Mean-encoding was first introduced by [catboost](#) from Yandex. It works best with categorical features. The most cited example is the [amazon](#) dataset challenge that consists only of categorical variables. The idea is to count the times a certain category level is associated with a positive target and compute the mean:

```
display(df1, df2)
```

	category	y
0	A	1
1	A	0
2	A	1
3	A	0
4	A	1
5	A	1
6	B	0
7	B	0
8	B	0
9	C	1
10	C	1
11	C	1
12	C	0

	category	mean_encoded
0	A	0.66
1	B	0.00
2	C	0.75

	category	y	mean_encoded
0	A	1	0.66
1	A	0	0.66
2	A	1	0.66
3	A	0	0.66
4	A	1	0.66
5	A	1	0.66
6	B	0	0.00
7	B	0	0.00
8	B	0	0.00
9	C	1	0.75
10	C	1	0.75
11	C	1	0.75
12	C	0	0.75

For the category level **A** we have:

$$\frac{(1+0+1+0+1+1)}{6} = 0.666$$

For the category level **C** we have:

$$\frac{(1+1+1+0)}{4} = 0.75$$

11.2 Some Literature

Important: if you're not allowed to read any more *towardsdatascience* article or *medium* articles – just remove the cookies for the page (inspect -> applications -> cookies) and reload the page afterwards. You can also open the page in *incognito-mode*.

- [Overfitting in Scaling prior to Hyperparameter Tuning and](#)
- [Benchmark of different target \(aka mean\) encoding strategies](#)
- [Validation Schemes](#)
- [nested cross-validation python-code-example](#)
- [Lessons Learned from Kaggle-Competitions](#)

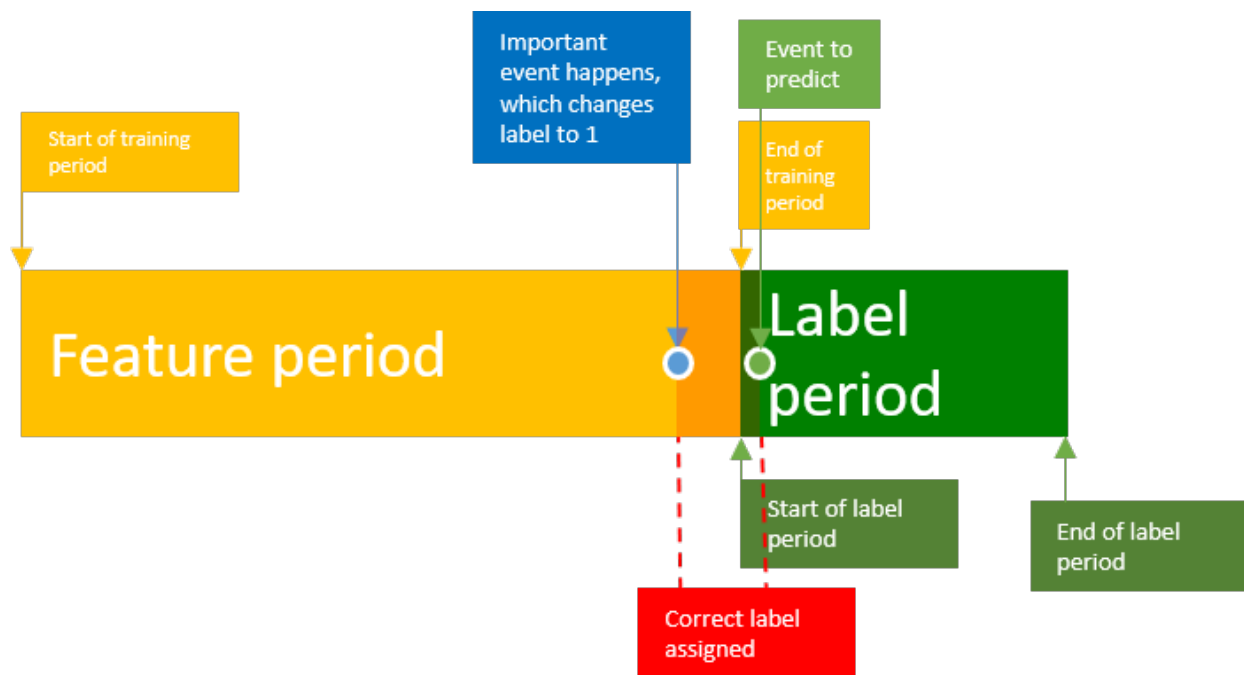
Part IV

Data Leakage and Data Dependence

DATE LEAKAGE AND DEPENDENT DATA

This topic is also highly relevant for the practical part of your grades. Please do have a look here: [How to avoid machine learning pitfalls: a guide for academic researchers](#).

In the picture below, you see the yellow time period called 'feature period' and the green time period, 'label period'. In the feature period data is collected, aggregated and transformed into 'features'. This data is the \mathbf{X} -matrix of variables. In the label period we are collecting the target, or \mathbf{y} -vector. The target \mathbf{y} is to be predicted by the help of \mathbf{X} . Data Leakage occurs when there is an important event (data point) in the feature period that already anticipates the target in an inadmissible way.




SOURCES OF DATA LEAKAGE

13.1 train data contains features that are not available in production

e.g., the row-number contains information about the target: first come the negative examples, the positive cases were then simply inserted underneath.

13.2 future data somehow slipped into the training set

e.g. Giba's property: [taken from kaggle](#)



Giba
1st place

The Data "Property"

Posted in [santander-value-prediction-challenge](#) 3 years ago

293

Hello everybody,
Since organizers officially declared that exploring the data property is good enough for the top solutions I decided to share my finding with the community to make that game fair to all participants while is time.

The dataset IS a time series in both dimensions, row wise and column wise :-)
And the column "f190486d6" is the last time stamp present in the dataset. So the target is 2 steps in the future of "f190486d6".

Look that example: [kernel](#)

This is one of the property I found. But I believe it can contain more.

Giba

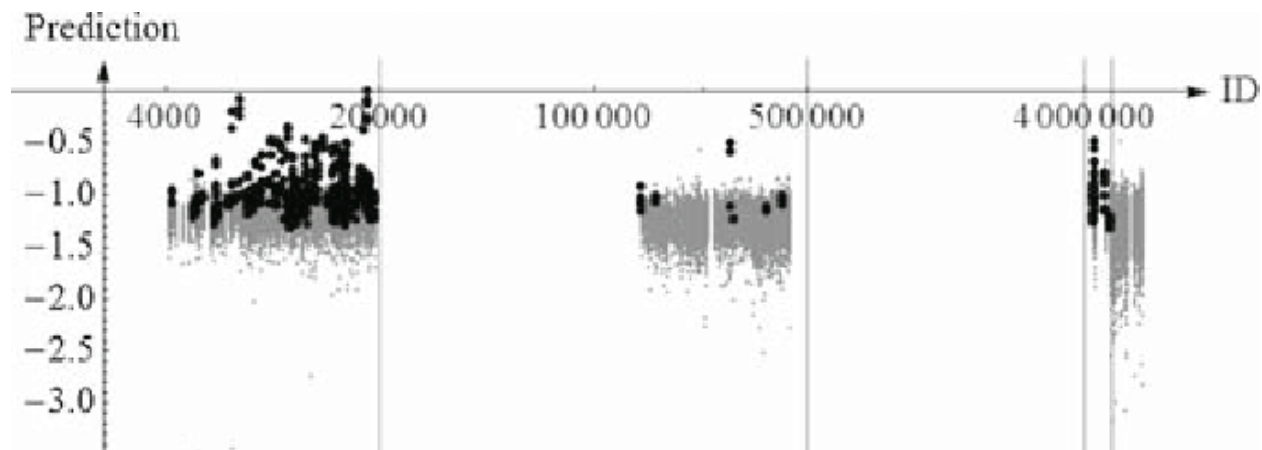
[Quote](#) | [Follow](#) | [Report](#) | [293 Upvoters](#)

and here is the mentioned data-structure: [this kernel](#) exploits the leakage

ID	target	f190486d6	58e2e02e6	eeb9cd3aa	9fd594eec	6eef030c1	15ace8c9f	fb0f5dbfe	58e05
7862786dc	3513333.3	0	1477600	1586889	75000	3147200	466461.5	1600000.0	0.0
c95732596	160000.0	310000	0	1477600	1586889	75000	3147200.0	466461.5	16000
16a02e67a	2352551.7	3513333	310000	0	1477600	1586889	75000.0	3147200.0	46646
ad960f947	280000.0	160000	3513333	310000	0	1477600	1586888.9	75000.0	31472
8adafb52	5450500.0	2352552	160000	3513333	310000	0	1477600.0	1586888.9	75000
fd0c7cfc2	1359000.0	280000	2352552	160000	3513333	310000	0.0	1477600.0	15868
a36b78ff7	60000.0	5450500	280000	2352552	160000	3513333	310000.0	0.0	14776
e42aae1b8	12000000.0	1359000	5450500	280000	2352552	160000	3513333.3	310000.0	0.0
0b132f2c6	500000.0	60000	1359000	5450500	280000	2352552	160000.0	3513333.3	31000
448efbb28	1878571.4	12000000	60000	1359000	5450500	280000	2352551.7	160000.0	35133
ca98b17ca	814800.0	500000	12000000	60000	1359000	5450500	280000.0	2352551.7	16000
2e57ec99f	307000.0	1878571	500000	12000000	60000	1359000	5450500.0	280000.0	23525
fef33cb02	528666.7	814800	1878571	500000	12000000	60000	1359000.0	5450500.0	28000

13.3 there is one feature that interacts with the target

taken from [Breast Cancer Identification: KDD CUP Winner's Report](#) Distribution of malignant (black) and benign (gray) candidates depending on patient ID on the X-axis in log scale.



13.4 Some more cases where we have data leakage:

- Customer advisor has a long call with customer and finally sells the product that is shipped only two weeks later. Variables 'last advisory contact' and 'length of call' certainly anticipate the product sale. When an algorithm learns to predict product propensity based on 'last advisor contact' it will ultimately suggest customers to the advisors for whom the advisor has already closed the deal.
- Train and test data is normalized with common sample statistics belonging to the whole data set
 - target encoding is dangerous: we will talk about it later on

- stacking is dangerous: we will discuss this topic as well

13.5 credit card applications

example taken from [here](#)

- card: Dummy variable, 1 if application for credit card accepted, 0 if not
- reports: Number of major derogatory reports
- age: Age n years plus twelfths of a year
- income: Yearly income (divided by 10,000)
- share: Ratio of monthly credit card expenditure to yearly income
- expenditure: Average monthly credit card expenditure
- owner: 1 if owns their home, 0 if rent
- selfempl: 1 if self employed, 0 if not.
- dependents: 1 + number of dependents
- months: Months living at current address
- majorcards: Number of major credit cards held
- active: Number of active credit accounts

```
import pandas as pd

url = 'https://raw.githubusercontent.com/YoshiKitaguchi/Credit-card-verification-
project/master/AER_credit_card_data.csv'
df = pd.read_csv(url, error_bad_lines=False, true_values = ['yes'], false_values = [
'no'])
print(df.head())
```

	card	reports	age	income	share	expenditure	owner	selfemp	\
0	True	0	37.66667	4.5200	0.033270	124.983300	True	False	
1	True	0	33.25000	2.4200	0.005217	9.854167	False	False	
2	True	0	33.66667	4.5000	0.004156	15.000000	True	False	
3	True	0	30.50000	2.5400	0.065214	137.869200	False	False	
4	True	0	32.16667	9.7867	0.067051	546.503300	True	False	

	dependents	months	majorcards	active
0	3	54	1	12
1	3	34	1	13
2	4	58	1	5
3	0	25	1	7
4	2	64	1	5

```
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
import numpy as np
import lightgbm

y = df['card']
X = df.drop('card', axis=1)
```

```

model = lightgbm.LGBMClassifier(boosting_type='gbdt', num_leaves=31, max_depth=- 1,
    ↪learning_rate=0.1,
                                n_estimators=500, subsample_for_bin=20000, objective=
    ↪'binary',
                                subsample=1.0, subsample_freq=0, colsample_bytree=1.0,
                                n_jobs=- 1, silent=True, importance_type='split',
                                is_unbalance = False, scale_pos_weight = 1.0)
model_pipe = make_pipeline(model)
cv_scores = cross_val_score(model_pipe, X, y, scoring='accuracy')
print(np.mean(cv_scores))

```

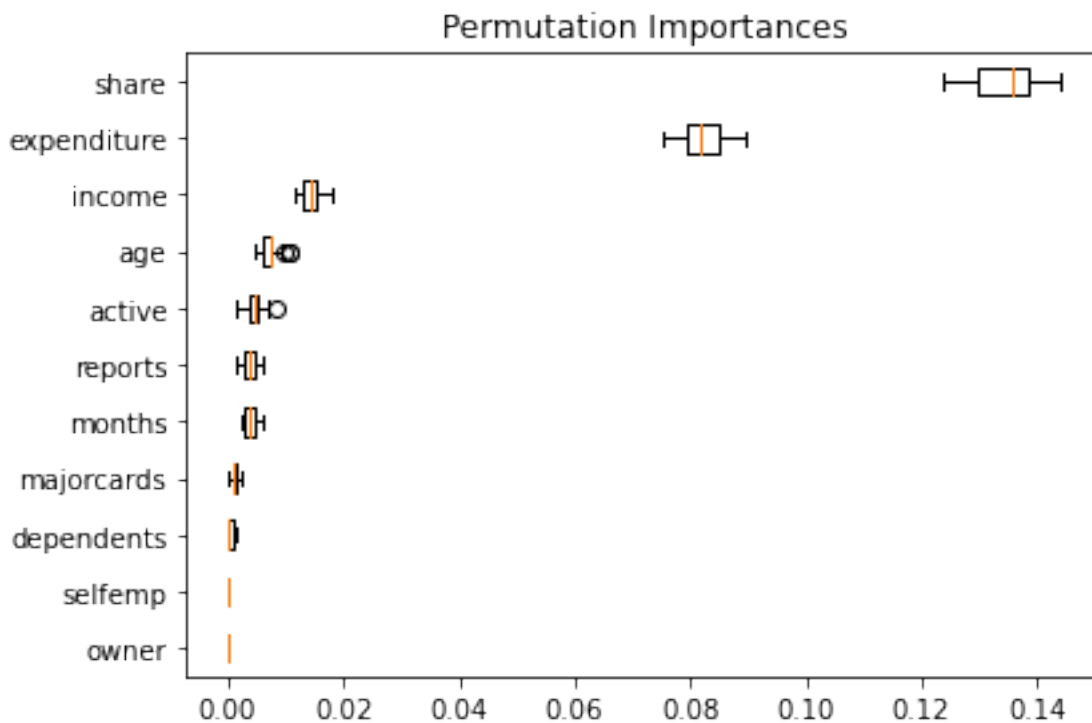
```
0.9765065099665862
```

```

from sklearn.inspection import permutation_importance
from matplotlib import pyplot as plt
model.fit(X, y)
result = permutation_importance(model, X, y,
    n_repeats=30,
    random_state=0)
sorted_idx = result.importances_mean.argsort()

fig, ax = plt.subplots()
ax.boxplot(result.importances[sorted_idx].T,
            vert=False, labels=X.columns[sorted_idx])
ax.set_title("Permutation Importances")
fig.tight_layout()
plt.show()

```

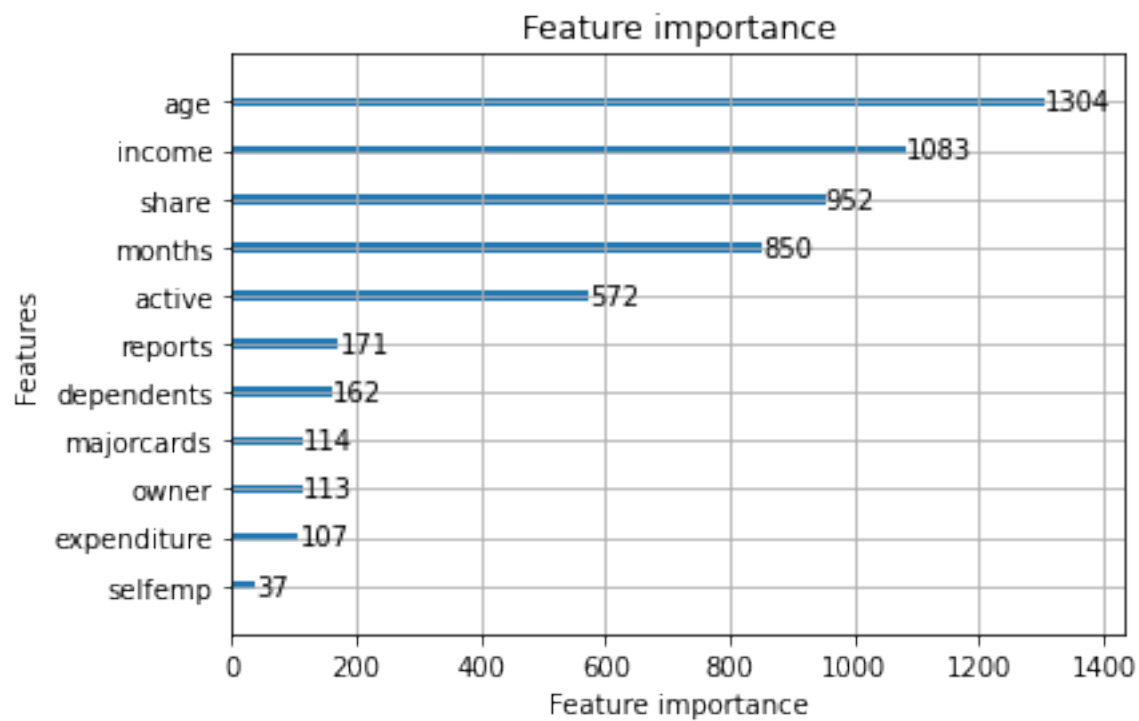


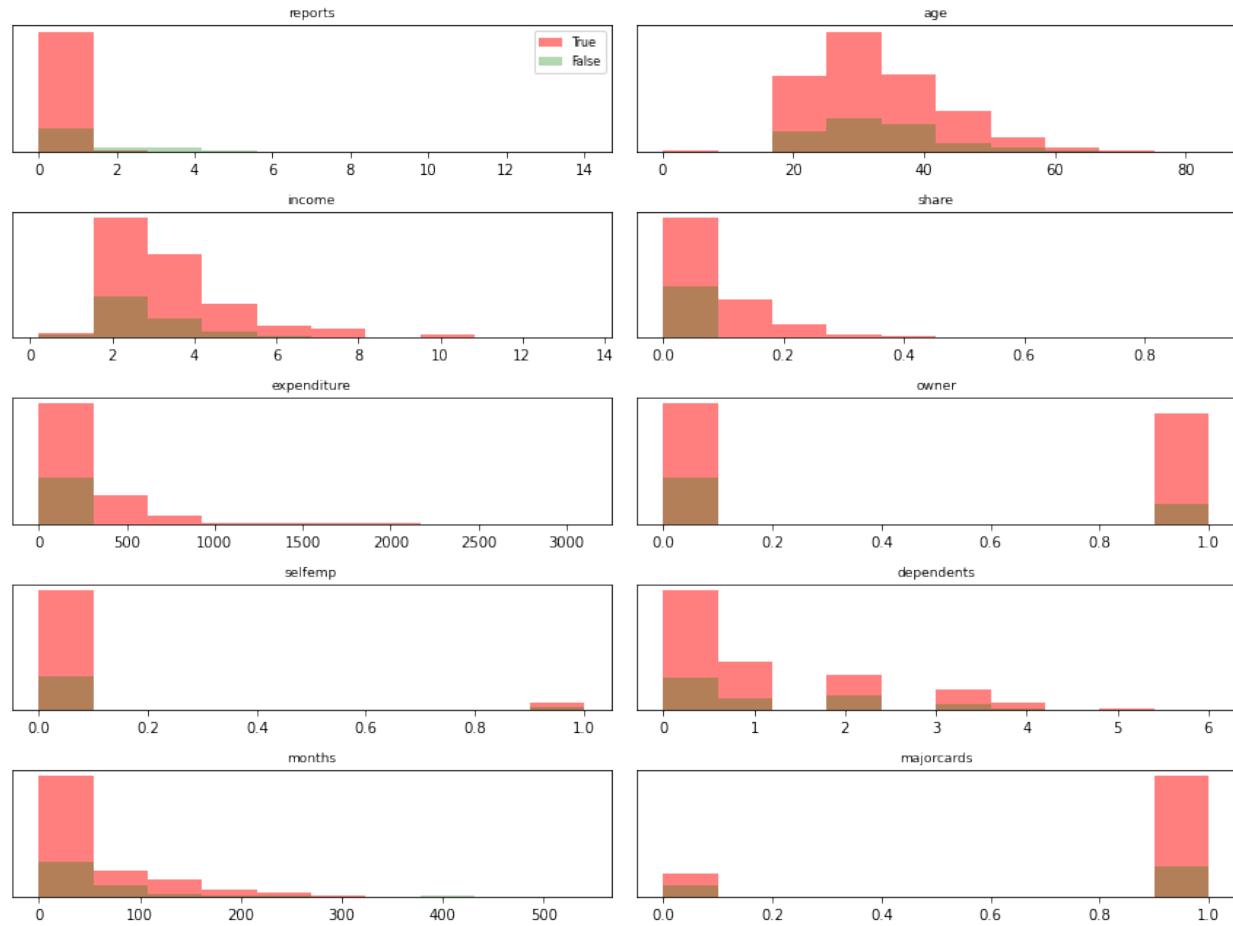
```

model.fit(X, y)
lightgbm.plot_importance(model)

```

```
<AxesSubplot:title={'center':'Feature importance'}, xlabel='Feature importance',  
ylabel='Features'>
```





```
display(X.loc[y == True, 'expenditure'].mean(), X.loc[y == False, 'expenditure'].
      ↪mean())
```

```
238.60242068103616
```

```
0.0
```

```
display(X.loc[y == True, 'share'].mean(), X.loc[y == False, 'share'].mean())
```

```
0.08848152972453567
```

```
0.0004767954841216091
```

```
from sklearn import tree
from dtreeviz.trees import *
import matplotlib.pyplot as plt
classifier = tree.DecisionTreeClassifier(max_depth=3) # limit depth of tree
classifier.fit(X, y)

viz = dtreeviz(classifier,
               X.values,
               y.values,
```

(continues on next page)

(continued from previous page)

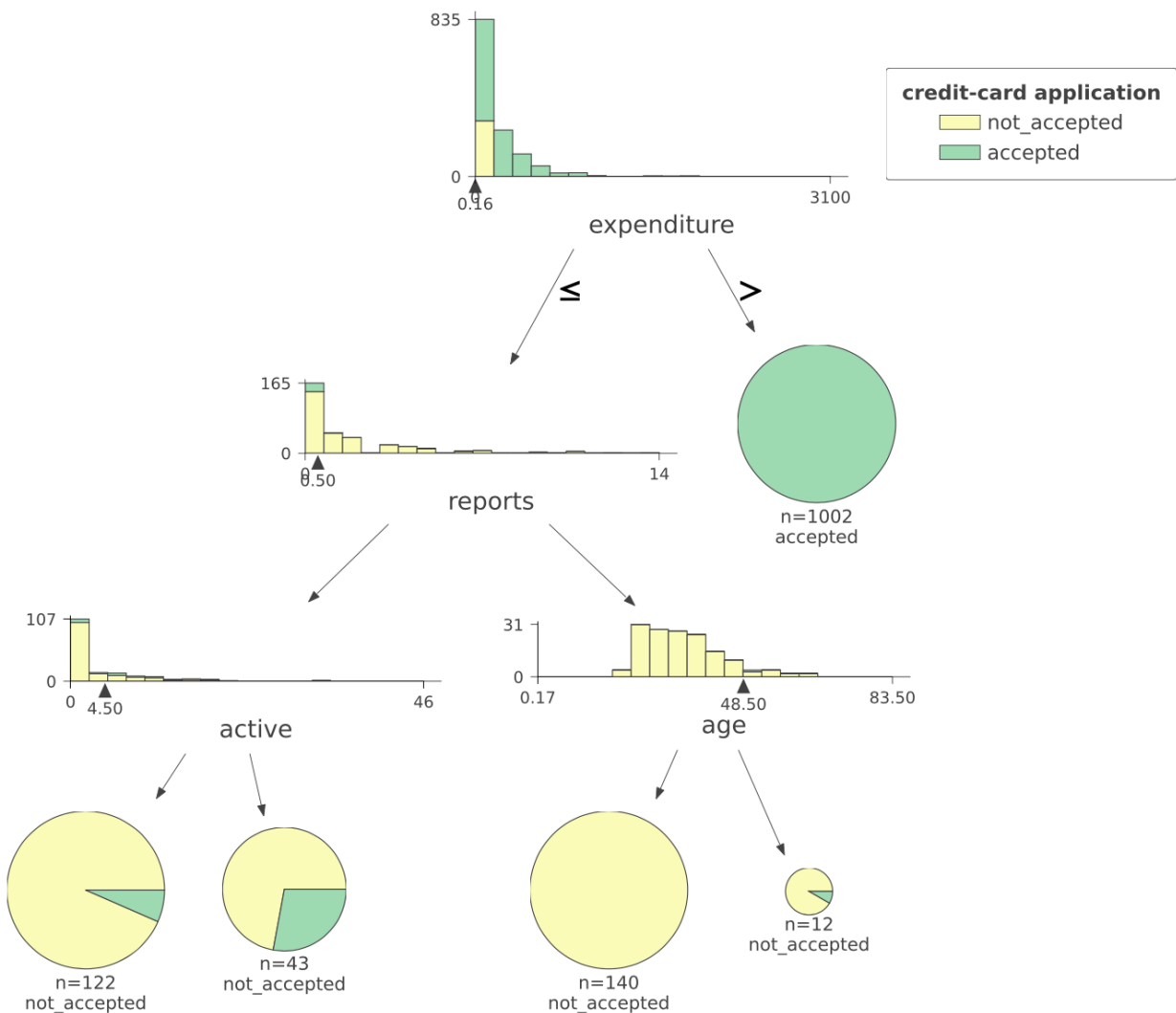
```

target_name='credit-card application',
feature_names=X.columns.tolist(),
class_names = ['not_accepted', 'accepted']
)

```

```
viz.save("decision_tree.svg")
```

```
<IPython.core.display.SVG object>
```



13.6 Solution:

Obviously,

- share: Ratio of monthly credit card expenditure to yearly income
- expenditure: Average monthly credit card expenditure

are features that suppose the applicant was granted a credit card.

DEPENDENCY BETWEEN DATA-SAMPLES

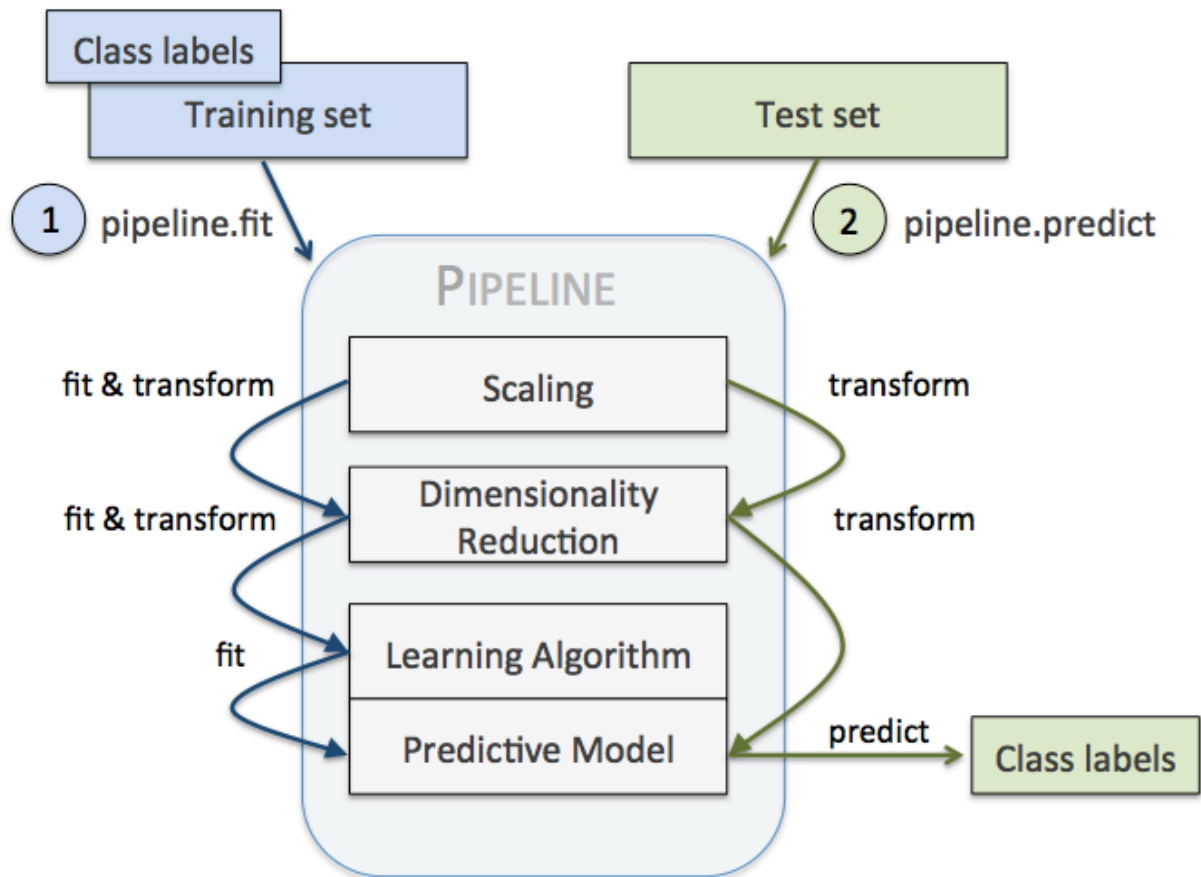
Training Machine Learning Algorithms works best, when we have many independent data samples in the training data. Dependent data arises when:

- we take repeatedly measures from the same individual (the trained algorithms will not generalize to other individuals)
- we take samples only from one bank (the socia-demographic structure of GKB's customers might be different from that of BCG's customers - as a result the algorithm will badly generalize)
-

Here, we are dealing with highly imbalanced data. One way to cope with it, is to replicate some examples of the minority-class (we will have a closer look at this problem in an extra notebook). By replicating some observations, the samples are not independent anymore.

14.1 First contact with Pipeline

`Pipeline` from `sklearn` is a tool that bundles different preprocessing steps. This is especially convenient when we are doing cross-validation; Without `Pipeline` we would have to code the different preprocessing steps as well as the cross-validation within a `for-loop`. The next graphic is taken from [here](#)



```
subsample_cc = pd.read_csv("../data/creditcard_subsampled.csv")
subsample_cc.head()
```

index	Time	V1	V2	V3	V4	V5	V6	\
0	541	406.0	-2.312227	1.951992	-1.609851	3.997906	-0.522188	-1.426545
1	623	472.0	-3.043541	-3.157307	1.088463	2.288644	1.359805	-1.064823
2	4920	4462.0	-2.303350	1.759247	-0.359745	2.330243	-0.821628	-0.075788
3	6108	6986.0	-4.397974	1.358367	-2.592844	2.679787	-1.128131	-1.706536
4	6329	7519.0	1.234235	3.019740	-4.304597	4.732795	3.624201	-1.357746

	V7	V8	...	V21	V22	V23	V24	V25	\
0	-2.537387	1.391657	...	0.517232	-0.035049	-0.465211	0.320198	0.044519	
1	0.325574	-0.067794	...	0.661696	0.435477	1.375966	-0.293803	0.279798	
2	0.562320	-0.399147	...	-0.294166	-0.932391	0.172726	-0.087330	-0.156114	
3	-3.496197	-0.248778	...	0.573574	0.176968	-0.436207	-0.053502	0.252405	
4	1.713445	-0.496358	...	-0.379068	-0.704181	-0.656805	-1.632653	1.488901	

	V26	V27	V28	Amount	Class
0	0.177840	0.261145	-0.143276	0.00	1
1	-0.145362	-0.252773	0.035764	529.00	1
2	-0.542628	0.039566	-0.153029	239.93	1
3	-0.657488	-0.827136	0.849573	59.00	1
4	0.566797	-0.010016	0.146793	1.00	1

[5 rows x 32 columns]

```
pd.crosstab(subsample_cc.Class.astype(str) + "_true", pd.Series(np.zeros_
↳like(subsample_cc.Class)).astype(str) + "_Actual")
```

```
col_0    0_Actual
Class
0_true    100000
1_true      492
```

One strategy in these cases is oversampling of the minority class:

```
import imblearn
from imblearn.over_sampling import SMOTE
y_train = subsample_cc.Class
X_train = subsample_cc.drop('Class', axis = 1)

# This is where we replicate some of the positive cases
X_train_ov, y_train_ov = SMOTE(sampling_strategy=0.03).fit_resample(X_train, y_train)
pd.crosstab(y_train_ov.astype(str) + "_true", pd.Series(np.zeros_like(y_train_ov)).
↳astype(str) + "_Actual")
```

```
col_0    0_Actual
Class
0_true    100000
1_true      3000
```

```
col_0    0.0_Actual
row_0
0.0_true    100000
1.0_true      1500
```

We take a gradient-boosting-classifier (more on this topic later) and compute the mean f1-score for the different folds:

```
from sklearn.metrics import f1_score, make_scorer
model = lightgbm.LGBMClassifier(boosting_type='gbdt', num_leaves=31, max_depth=- 1,
↳learning_rate=0.1,
                                n_estimators=500, subsample_for_bin=20000, objective=
↳'binary',
                                subsample=1.0, subsample_freq=0, colsample_bytree=1.0,
                                n_jobs=- 1, silent=True, importance_type='split',
                                is_unbalance = False, scale_pos_weight = 1.0)
model_pipe = make_pipeline(model)
cv_scores = cross_val_score(model_pipe, X_train_ov, y_train_ov, scoring=make_
↳scorer(f1_score, labels=[2]), cv = 2)
print(np.mean(cv_scores))
```

```
0.9828092562783878
```

what happens when we augment the number of cross-validation folds?

- we have more training-data in the folds we use for training
- the probability that some of the replicated cases are in the training and in the test set is higher -> more data leakage

```
folds = 5
cv_scores = cross_val_score(model_pipe, X_train_ov, y_train_ov, scoring=make_
↳scorer(f1_score, labels=[2]), cv = folds)
print(f"""f1-score for {folds}-fold cross-validation: {np.mean(cv_scores)}""")
```

(continues on next page)

(continued from previous page)

```

folds = 10
cv_scores = cross_val_score(model_pipe, X_train_ov, y_train_ov, scoring=make_
    ↳scorer(f1_score, labels=[2]), cv = folds)
print(f"""f1-score for {folds}-fold cross-validation: {np.mean(cv_scores)}""")

```

```

f1-score for 5-fold cross-validation: 0.9879242696875334
f1-score for 10-fold cross-validation: 0.9872838257834472

```

14.2 the correct way to do it: oversample within each fold:

This is where Pipelineshines

```

from imblearn.pipeline import Pipeline
ov_pipeline = Pipeline([
    ('sampling', SMOTE(sampling_strategy=0.03)),
    ('classification', model)
])
folds = 5
cv_scores = cross_val_score(ov_pipeline, X_train, y_train, scoring=make_scorer(f1_
    ↳score, labels=[2]), cv=folds)

print(f"""f1-score for {folds}-fold cross-validation: {np.mean(cv_scores)}""")
folds = 10
cv_scores = cross_val_score(ov_pipeline, X_train, y_train, scoring=make_scorer(f1_
    ↳score, labels=[2]), cv=folds)
print(f"""f1-score for {folds}-fold cross-validation: {np.mean(cv_scores)}""")

```

```

f1-score for 5-fold cross-validation: 0.788538141844491
f1-score for 10-fold cross-validation: 0.8614838865031157

```

Compare those results with the ones we obtained when we did first the up-sampling and then the cross-validation!

14.3 Some more cases where we have dependent data

- Repeatedly sampling data from the same individual:
 - Fraud: A fraudster commits many frauds that have a similar pattern; For example for every fraud committed, the fraudster uses a different account of the same bank in Thailand. When doing cross-validation we have frauds related to the very bank in Thailand in the training set as well as in the test set. Hence, we will overestimate the capability of the trained classifier to generalize to new, unseen fraud cases. But it will be very efficient to detect this one fraudster with bank accounts in Thailand.
 - customer journey: to detect an event as soon as possible, data is sampled with different offsets before the event's occurrence. When trying to predict an event we could be tempted to sample data from different points in time before the event. This data will always be very similar and is hence dependent. For example, medical health records are not changing very fast and blood pressure two months ago will be similar to that measured one month ago. Most bank accounts have a similar balance in a one months distance.
 - classifying websites: social media websites belong all to facebook. There are just not enough social media websites to learn something about them in the training set and generalize to other social media websites in the test set.

- Train and test data is normalized with common sample statistics belonging to the whole data set
 - target encoding is dangerous: we will talk about it later on
 - stacking is dangerous: we will discuss this topic as well
- Sentence Classification: sentences belonging to the same document
 - customer churn: An angry customer sends frequent e-mails. All e-mails happen to have the same characteristics, e.g. instead of the *Umlaut* ‘ü’, the customer uses ‘ue’. The algorithm might be tempted to learn that ‘ue’ is a special churn-characteristic. When half of the e-mails end up in the train set and the rest in the test set, we will overestimate the prediction accuracy of the learned algorithm.
 - When building a classifier to distinct medical publications from IT-related publications, it is important to have a representative sample of medical topics as well as tech-topics. When taking sentences from one document that is heavily Java related, the algorithm will struggle to generalize to the programming language Python. When the ‘Java-sentences’ are in the train set as well as in the test set, we will overestimate the algorithm’s performance on new documents.
- Diagnosis: patient records coming from the same hospital
 - Hospitals might have different specializations; When we want to predict diagnosis based on the doctors’ reports, cancer cases from a clinic specialized in cancer treatments might have higher similarity to each other than cancer cases coming from a orthopaedic hospital. When the reports of the specialized clinic end up in the train set as well as in the test set, we will overestimate the algorithm’s capability to correctly classify the diagnosis ‘cancer’.

14.3.1 recent article summarizing errors when predicting COVID:

Excerpts of the article [Hundreds of AI tools have been built to catch covid. None of them helped.](#):

“They looked at 415 published tools and, like Wynants and her colleagues, concluded that none were fit for clinical use.”

“Both teams found that researchers repeated the same basic errors in the way they trained or tested their tools.”

“Many of the problems that were uncovered are linked to the poor quality of the data that researchers used to develop their tools.”

- **duplicates:** “Driggs highlights the problem of what he calls Frankenstein data sets, which are spliced together from multiple sources and can contain duplicates.”
- **confounding variables:**
 - “Many unwittingly used a data set that contained chest scans of children who did not have covid as their examples of what non-covid cases looked like. But as a result, the AIs learned to identify kids, not covid.”
 - “Because patients scanned while lying down were more likely to be seriously ill, the AI learned wrongly to predict serious covid risk from a person’s position.”
- **different sources:** “In yet other cases, some AIs were found to be picking up on the text font that certain hospitals used to label the scans. As a result, fonts from hospitals with more serious caseloads became predictors of covid risk.”
- **human labeling error:** “It would be much better to label a medical scan with the result of a PCR test rather than one doctor’s opinion, says Driggs.”

14.3.2 How to fix it?

- “Better data would help, but in times of crisis that’s a big ask.”
- ““Until we buy into the idea that we need to sort out the unsexy problems before the sexy ones, we’re doomed to repeat the same mistakes,” says Mateen.”

Original articles: Wynants et al., 2020. Prediction models for diagnosis and prognosis of covid-19: systematic review and critical appraisal Roberts et al., 2021. Common pitfalls and recommendations for using machine learning to detect and prognosticate for COVID-19 using chest radiographs and CT scans

14.3.3 Data leakage Literature

- examples of data leakage in competitions: start on page 19
- alternative to the text above here is the video
- Medical data mining: insights from winning two competitions

important: if you’re not allowed to read any more ‘towardsdatascience’ article or ‘medium’ articles – just remove the cookies for the page (inspect -> applications -> cookies) and reload the page afterwards.

- data leakage when tuning hyper-parameters