Fachhochschule Nordwestschweiz
Hochschule für Technik

Machine Learning

FS2019

Lerneinheit 3

Bias-Variance-Trade-Off, Regularisierung

Dozent: Michael Graber
michael.graber@fhnw.ch

Diese Vorlesungsunterlagen sind aus den Folien der Vorlesungen zusammengestellt worden.

Lernziele

- Sie fühlen sich sicher mit (multipler) linearer Regression.
- Sie kennen 3 verschiedene Varianten, um mit *Least Squares* die Koeffizienten der linearen Regression zu bestimmen.
- Sie kennen Methoden, um kategorische Variablen in Regressions-Modellen verwenden zu können.
- Sie wissen wie nicht-lineare Effekte mit linearer Regression modelliert werden können.
- Sie verstehen den *Bias-Variance Trade-off* und das Problem von *Overfitting*.
- Sie kennen erste Ansätze für *Model Selection*.
- Sie kennen *Ridge Regression* und wissen wie diese implementiert werden kann, sowie deren Eigenschaften.
- Sie kennen den Unterschied von *Lasso* und *Ridge Regression* und können deren Koeffizienten-'Pfade' interpretieren.

1 Lineare Regression Recap

$$y = \mathbf{X}\beta + \varepsilon \tag{1}$$

$$l(\beta, x^{(i)}) = x^{(i)}\beta = \hat{y}^{(i)} \tag{2}$$

$$J(\beta) = \sum_{i=1}^{N} (y^{(i)} - l(\beta, x^{(i)}))^2 \tag{3}$$

Least Squares

$$\hat{\beta} = \text{argmin}_\beta \quad J(\beta) = \text{argmin}_\beta \quad \sum_{i=1}^{N} (y^{(i)} - l(\beta, x^{(i)}))^2 \tag{4}$$

Lösungsansätze:

1. Gradient Descent

- fixed step size, line search
- Parameter : `stepsize`, `alpha`, `beta`, `epsilon`, ..

- Wie untersucht man die Konvergenz des Verfahrens?

  Ableitung am Punkt x immer näher an 0 annähert?

2. Normalengleichung

Die Normalengleichung ist eine analytische Lösung des *Least Squares* Ansatzes der linearen Regression.

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{5}$$

3. library code!

Es besteht aber auch die Möglichkeit, dass sie Bibliotheks-Funktionen verwenden, um den *'Fit'* der Modell-Koeffizienten β vorzunehmen.

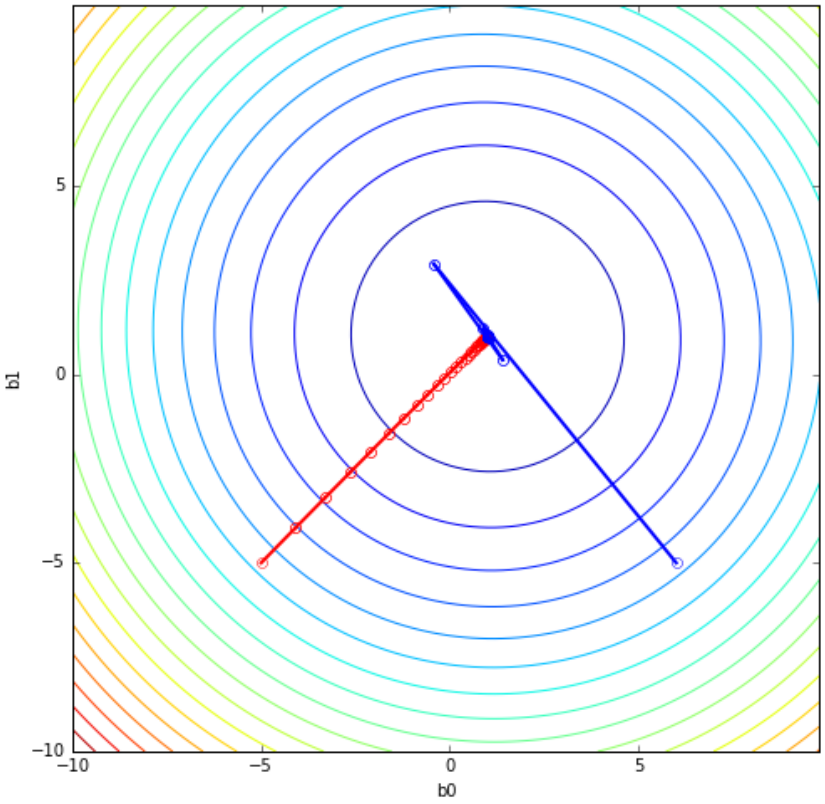
In `numpy` gibt es eine entsprechende Funktion:

```
In [1]: import numpy as np
```

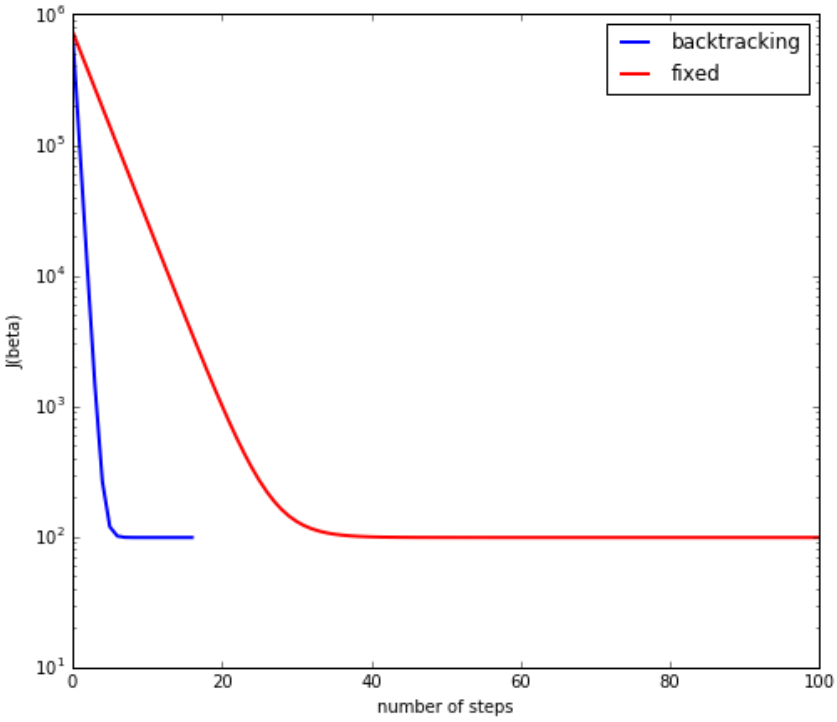
```
In [2]: print(np.linalg.lstsq.__doc__)
```

```
    Return the least-squares solution to a linear matrix equation.

    Solves the equation `a x = b` by computing a vector `x` that
    minimizes the Euclidean 2-norm `|| b - a x ||^2`.  The equation may
```



Contour Plot and Gradient Descent Paths



Learning Curves

```
be under-, well-, or over- determined (i.e., the number of
linearly independent rows of `a` can be less than, equal to, or
greater than its number of linearly independent columns).  If `a`
is square and of full rank, then `x` (but for round-off error) is
the "exact" solution of the equation.

Parameters
----------
a : (M, N) array_like
    "Coefficient" matrix.
b : {(M,), (M, K)} array_like
    Ordinate or "dependent variable" values. If `b` is two-dimensional,
    the least-squares solution is calculated for each of the `K` columns
    of `b`.
rcond : float, optional
    Cut-off ratio for small singular values of `a`.
    For the purposes of rank determination, singular values are treated
    as zero if they are smaller than `rcond` times the largest singular
    value of `a`.

Returns
-------
x : {(N,), (N, K)} ndarray
```

```
        Least-squares solution. If `b` is two-dimensional,
        the solutions are in the `K` columns of `x`.
    residuals : {(), (1,), (K,)} ndarray
        Sums of residuals; squared Euclidean 2-norm for each column in
        ``b - a*x``.
        If the rank of `a` is < N or M <= N, this is an empty array.
        If `b` is 1-dimensional, this is a (1,) shape array.
        Otherwise the shape is (K,).
    rank : int
        Rank of matrix `a`.
    s : (min(M, N),) ndarray
        Singular values of `a`.


    ...
```

Wiederholung : Bestimmheitsmass

Anteil der Varianz der Output-Variablen welche durch das Modell erklärt wird:

$$R^2 = 1 - \frac{Var(r)}{Var(y)} = 1 - \frac{Var(y - \hat{y})}{Var(y - \bar{y})} \tag{6}$$

2 Mehrdimensionales Anwendungsbeispiel : Diabetes dataset

Laden von Daten mit pandas

Wir können die python Bibliothek `pandas` verwenden, um verschiedene Speicherformate für Datensätze in *Dataframes* zu laden und um grundlegende *Data Wrangling*-, Statistik- und Visualisierungs-Schritte darauf auszuführen:

```
In [3]: import pandas as pd
```

```
In [4]: df = pd.read_csv('/data/diabetes_data.csv')
```

```
In [5]: type(df)
```

```
Out[5]: pandas.core.frame.DataFrame
```

```
In [6]: df.head()
```

Out[6]:

	AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
0	59	2	32.1	101.0	157	93.2	38.0	4.0	4.8598	87	151
1	48	1	21.6	87.0	183	103.2	70.0	3.0	3.8918	69	75
2	72	2	30.5	93.0	156	93.6	41.0	4.0	4.6728	85	141
3	24	1	25.3	84.0	198	131.4	40.0	5.0	4.8903	89	206
4	50	1	23.0	101.0	192	125.4	52.0	4.0	4.2905	80	135

In [7]:
```
df.describe()
```

Out[7]:

	AGE	SEX	BMI	BP	S1	S2	\
count	442.000000	442.000000	442.000000	442.000000	442.000000	442.000000	
mean	48.518100	1.468326	26.375792	94.647014	189.140271	115.439140	
std	13.109028	0.499561	4.418122	13.831283	34.608052	30.413081	
min	19.000000	1.000000	18.000000	62.000000	97.000000	41.600000	
25%	38.250000	1.000000	23.200000	84.000000	164.250000	96.050000	
50%	50.000000	1.000000	25.700000	93.000000	186.000000	113.000000	
75%	59.000000	2.000000	29.275000	105.000000	209.750000	134.500000	
max	79.000000	2.000000	42.200000	133.000000	301.000000	242.400000	

	S3	S4	S5	S6	Y
count	442.000000	442.000000	442.000000	442.000000	442.000000
mean	49.788462	4.070249	4.641411	91.260181	152.133484
std	12.934202	1.290450	0.522391	11.496335	77.093005
min	22.000000	2.000000	3.258100	58.000000	25.000000
25%	40.250000	3.000000	4.276700	83.250000	87.000000
50%	48.000000	4.000000	4.620050	91.000000	140.500000
75%	57.750000	5.000000	4.997200	98.000000	211.500000
max	99.000000	9.090000	6.107000	124.000000	346.000000

In [8]:
```
df.dtypes
```

Out[8]:
```
AGE      int64
SEX      int64
BMI    float64
BP     float64
S1       int64
S2     float64
S3     float64
S4     float64
S5     float64
S6       int64
Y        int64
dtype: object
```

```
In [9]: # we are setting, arbitrarily, 1 = 'f'
        df.loc[df.SEX == 1, 'SEX'] = 'f'
        df.loc[df.SEX == 2, 'SEX'] = 'm'

        # and make a categorical variable out of SEX
        df['SEX'] = df['SEX'].astype('category')
```

```
In [10]: df.head()
```

Out[10]:

	AGE	SEX	BMI	BP	S1	S2	S3	S4	S5	S6	Y
0	59	m	32.1	101.0	157	93.2	38.0	4.0	4.8598	87	151
1	48	f	21.6	87.0	183	103.2	70.0	3.0	3.8918	69	75
2	72	m	30.5	93.0	156	93.6	41.0	4.0	4.6728	85	141
3	24	f	25.3	84.0	198	131.4	40.0	5.0	4.8903	89	206
4	50	f	23.0	101.0	192	125.4	52.0	4.0	4.2905	80	135

```
In [11]: df.dtypes
```

```
Out[11]: AGE       int64
         SEX    category
         BMI     float64
         BP      float64
         S1        int64
         S2      float64
         S3      float64
         S4      float64
         S5      float64
         S6        int64
         Y         int64
         dtype: object
```

pairplot mit seaborn

Die python Bibliothek seaborn bietet zahlreiche Erweiterungen von matplotlib für statistische Plots:

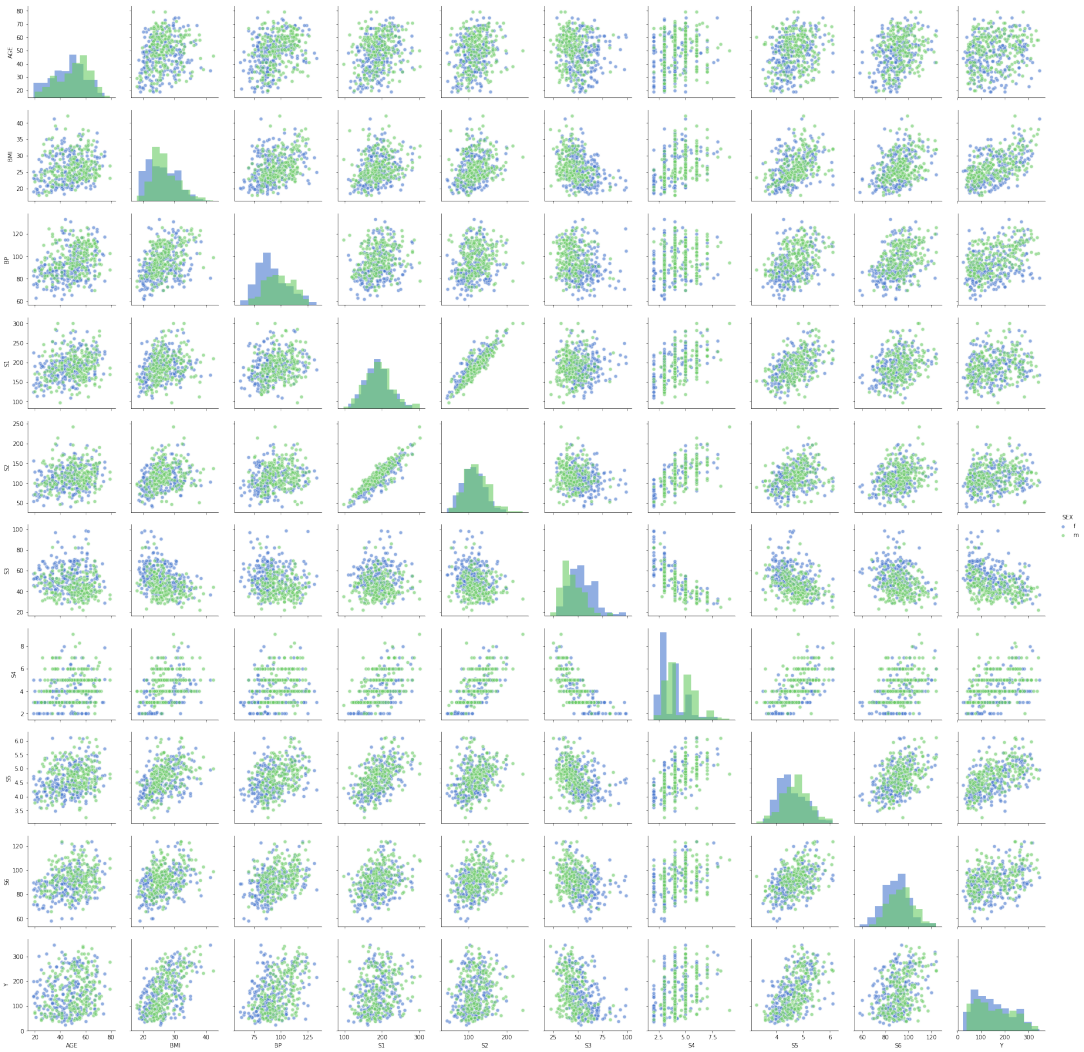
```
In [12]: from matplotlib import pyplot as plt
         %matplotlib inline

         import seaborn as sns
         plt.style.use('seaborn-muted')
```

Mit einem *Pairplot* können sie einfach einen ersten Eindruck der Verteilung der Datenpunkte und der Korrelationen von Variablen-Paaren gewinnen:

```
In [13]: # selecting some columns
         cols = df.columns.tolist()
         cols.remove('SEX')


         ALPHA = 0.6
         _ = sns.pairplot(df, vars=cols, hue='SEX',
                          plot_kws={'alpha': ALPHA},
                          diag_kind='hist', diag_kws={ 'alpha': ALPHA }
                         )
```



Im Kontext von (linearen) Regressionproblemen interessieren uns **funktionale**, insbesondere lineare, **Zusammenhänge** zwischen der abhängigen Variablen und den unabhängigen Variablen. Wie wir etwas später sehen

werden sind auch funktionale Zusammenhänge zwischen abhängigen Variablen wichtig.

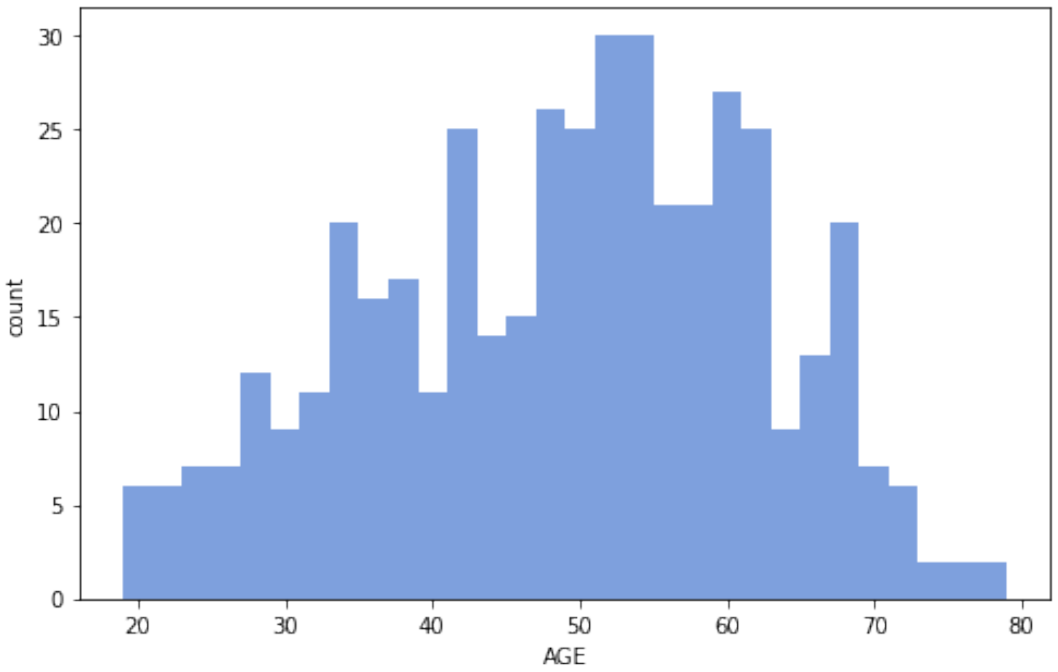
Einen visuellen Eindruck funktionaler Zusammenhänge können wir mit dem *Pairplot* einfach gewinnen.

Histogramm Recap

Ein *Histogramm* erlaubt Einsicht in die Verteilung der Werte einzelner Variablen.

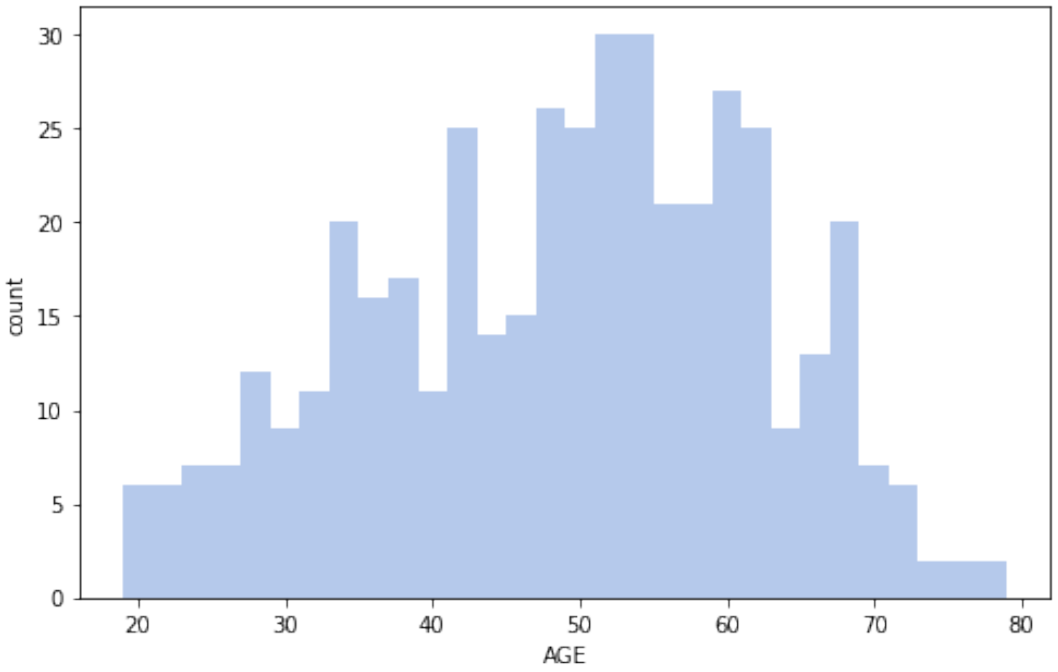
Mit `matplotlib` lässt sich ein Histogramm einfach erstellen:

```
In [14]: # create figure an axes with specific size
         fig, ax = plt.subplots(figsize=(8, 5))
         # plot into axes
         _ = ax.hist(np.array(df['AGE']), bins=30, alpha=0.7)
         # label axes
         _ = ax.set_xlabel('AGE')
         _ = ax.set_ylabel('count')
```



Eine weitere Möglichkeit ist die Verwendung von `seaborn`:

```
In [15]: fig, ax = plt.subplots(figsize=(8, 5))
         _ = sns.distplot(df.AGE, kde=False, bins=30)
         _ = ax.set_ylabel('count')
```



Wählen Sie die Anzahl *Bins* sinnvoll, sodass Sie einen akkuraten Eindruck der Verteilung der Variablen bekommen können! -> siehe DSP.

Lineare Regression mit scikit-learn

```
In [16]: from sklearn.linear_model import LinearRegression
```

Alle Variablen mit Namen.

```
In [17]: df.columns
```

```
Out[17]: Index(['AGE', 'SEX', 'BMI', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6', 'Y'], dtype='object')
```

Wir extrahieren *Input* und *Output* und erstellen davon `numpy`-Arrays. Vorerst noch ohne die Variable `SEX`.

```
In [18]: X = np.array(df[['AGE', 'BMI', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6',]])
         y = np.array(df['Y'])
```

```
In [19]: X
```

```
Out[19]: array([[  59.    ,   32.1   ,  101.    , ...,    4.    ,    4.8598,   87.    ],
                [  48.    ,   21.6   ,   87.    , ...,    3.    ,    3.8918,   69.    ],
                [  72.    ,   30.5   ,   93.    , ...,    4.    ,    4.6728,   85.    ],
                ...,
                [  60.    ,   24.9   ,   99.67  , ...,    3.77  ,    4.1271,   95.    ],
                [  36.    ,   30.    ,   95.    , ...,    4.79  ,    5.1299,   85.    ],
                [  36.    ,   19.6   ,   71.    , ...,    3.    ,    4.5951,   92.    ]])
```

```
In [20]: y
```

```
Out[20]: array([151,  75, 141, 206, 135,  97, 138,  63, 110, 310, 101,  69, 179,
                ...
                94, 183,  66, 173,  72,  49,  64,  48, 178, 104, 132, 220,  57])
```

Nun initialisieren wir einen `LinearRegression`-*Estimator* und *fitten* die Modell-Koeffizienten:

```
In [21]: lr = LinearRegression()
         lr.fit(X, y)
```

```
Out[21]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Bestimmtheitsmass R^2

```
In [22]: lr.score(X, y)
```

```
Out[22]: 0.50057995591251769
```

β_0

```
In [23]: lr.intercept_
```

```
Out[23]: -363.898716034671
```

β

```
In [24]: lr.coef_
```

```
Out[24]: array([ -0.12051511,   6.00406612,   0.95050794,  -0.98078427,
                  0.65849619,   0.51362821,   4.65988116,  68.9473421 ,   0.2026253 ])
```

Daten 'prä-prozessieren'

Machine Learning Algorithmen operieren schlussendlich immer mit numerischen Variablen. Kategorische oder ordinale Variablen liegen oftmals aber gar nicht in numerischer Form vor. Sie müssen erst 'prä-prozessiert' werden.

Dazu gibt es weitere Methoden Daten zu 'prä-prozessieren'. Diese 'Prä-Prozessierungs'-Schritte haben manchmal einen Einfluss auf das Machine Learning Resultat oder das Konvergenz-Verhalten des Optimierungs-Algorithmus.

Wir werden uns nun einige dieser Methoden anschauen:

One-Hot Encoding

Die Variable `SEX` ist eine kategorische Variable:

```
In [25]: df['SEX'].values
```

```
Out[25]: [m, f, m, f, f, ..., m, m, m, f, f]
         Length: 442
         Categories (2, object): [f, m]
```

Wir wissen hier nicht ob '1' oder '2' für 'männlich' oder 'weiblich steht. Das spielt aber auch keine Rolle. Wichtig ist, dass wir diese Variable nicht dirket als numerische Variable für lineare Regression verwenden können. Wir müssen sie erst in eine 'brauchbare' numerische Variable umwandeln. Ein Ansatz dazu bietet '*One-Hot-Encoding*':

```
In [26]: male = df['SEX'] == 'm'
         male = np.array(male).astype('int')
         male = np.expand_dims(male, axis=1)
         male.T
```

```
Out[26]: array([[1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1,
                 ...
                 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1,
                 0, 0]])
```

```
In [27]: female = df['SEX'] == 'f'
         female = np.array(female).astype('int')
         female = np.expand_dims(female, axis=1)
         female.T
```

```
Out[27]: array([[0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0,
                 ...
                 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0,
                 1, 1]])
```

Im Fall von zwei komplementären Kategorien reicht es, die Indikator-Variable der einen Kategorie den abhängigen Variablen X zuzufügen:

```
In [28]: X = np.hstack((X, female))
```

```
In [29]: LinearRegression().fit(X, y).score(X, y)
```

```
Out[29]: 0.51774842222034978
```

`scikit-learn` preprocessing tools

`scikit-learn` verfügt auch über eine Reihe von *'preprocessing tools'*. Sie finden diese unter `sklearn.preprocessing`. Hier gibt es auch einen `OneHotEncoder`.

```
In [30]: from sklearn.preprocessing import OneHotEncoder, LabelEncoder
```

```
In [31]: # up to sklearn 0.20 we need to first make integer labels of categoricals given
         as strings
         lenc = LabelEncoder().fit(df['SEX'])
         intlabels = lenc.transform(df['SEX'])

         lenc.classes_
```

```
Out[31]: array(['f', 'm'], dtype=object)
```

```
In [32]: intlabels
```

```
Out[32]: array([1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0,
                ...
                1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0,
                1, 1, 1, 0, 0])
```

```
In [33]: # starting from sklearn 0.20 OneHotEncoder could take categorical string
         variables directly
         ohe = OneHotEncoder().fit(np.atleast_2d(intlabels).T)

         ohe.active_features_
```

```
Out[33]: array([0, 1])
```

```
In [34]: ohevar = ohe.transform(np.atleast_2d(intlabels).T)
```

Wir kriegen eine Variable in 'sparsem' Datenformat zurück:

```
In [35]: ohevar
```

```
Out[35]: <442x2 sparse matrix of type '<class 'numpy.float64'>'
                 with 442 stored elements in Compressed Sparse Row format>
```

```
In [36]: ohevar.toarray()
```

```
Out[36]: array([[ 0.,  1.],
                [ 1.,  0.],
                [ 0.,  1.],
                [ 1.,  0.],
                ...
                [ 0.,  1.],
                [ 1.,  0.],
                [ 1.,  0.]])
```

Diese One-Hot-encodierten Variablen können wir nun analog wie oben X hinzufügen.

Zusätzliche Variablen durch *Variablen-Transformation*

Schlussendlich können wir auf unseren ursprünglich 'gemessenen' Variablen beliebige Transformationen ausführen um 'neue', zusätzliche Variablen / *Features* zu erzeugen. Nehmen wir an, wir hätten mit jedem Datenpunkt 3 Variablen gemessen : $x^{(i)} \in \mathbf{R}^3$, also $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)})$. Der Lesbarkeit halber lassen wir nun den Index für den Datenpunkt weg, also $x = (x_1, x_2, x_3)$.

Wir können nun zu den bereits existierenden Variablen zum Beispiel Variablen der Quadrate der existierenden Funktionen hinzufügen:

$$x = (x_1, x_2, x_3, x_1^2, x_2^2, x_3^2) \tag{7}$$

.. oder Interaktionsterme mehrerer Variablen:

$$x = (x_1, x_2, x_3, x_1 \cdot x_2, x_1 \cdot x_3, x_2 \cdot x_3) \tag{8}$$

.. oder gar beides

```
In [37]: from sklearn.preprocessing import PolynomialFeatures
```

```
In [38]: print(PolynomialFeatures.__doc__)
```

```
Generate polynomial and interaction features.

    Generate a new feature matrix consisting of all polynomial combinations
    of the features with degree less than or equal to the specified degree.
    For example, if an input sample is two dimensional and of the form
    [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

    Parameters
    ----------
    degree : integer
        The degree of the polynomial features. Default = 2.

    interaction_only : boolean, default = False
        If true, only interaction features are produced: features that are
        products of at most ``degree`` *distinct* input features (so not
        ``x[1] ** 2``, ``x[0] * x[2] ** 3``, etc.).

    include_bias : boolean
        If True (default), then include a bias column, the feature in which
        all polynomial powers are zero (i.e. a column of ones - acts as an
        intercept term in a linear model).

    Examples
    --------
    >>> X = np.arange(6).reshape(3, 2)
    >>> X
    array([[0, 1],
           [2, 3],
           [4, 5]])
    >>> poly = PolynomialFeatures(2)
    >>> poly.fit_transform(X)
    array([[  1.,   0.,   1.,   0.,   0.,   1.],
           [  1.,   2.,   3.,   4.,   6.,   9.],
           [  1.,   4.,   5.,  16.,  20.,  25.]])
    >>> poly = PolynomialFeatures(interaction_only=True)
    >>> poly.fit_transform(X)
    array([[  1.,   0.,   1.,   0.],
           [  1.,   2.,   3.,   6.],
```

```
        [  1.,   4.,   5.,  20.]])

    Attributes
    ----------
    powers_ : array, shape (n_output_features, n_input_features)
        powers_[i, j] is the exponent of the jth input in the ith output.

    n_input_features_ : int
        The total number of input features.

    n_output_features_ : int
        The total number of polynomial output features. The number of output
        features is computed by iterating over all suitably sized combinations
        of input features.

    Notes
    -----
    Be aware that the number of features in the output array scales
    polynomially in the number of features of the input array, and
    exponentially in the degree. High degrees can cause overfitting.

    See :ref:`examples/linear_model/plot_polynomial_interpolation.py
    <sphx_glr_auto_examples_linear_model_plot_polynomial_interpolation.py>`
```

```
In [39]: PolynomialFeatures(2).fit_transform(
             np.array([
                 [2, 3, 4],
                 [5, 6, 7]
             ])
         )
```

```
Out[39]: array([[  1.,   2.,   3.,   4.,   4.,   6.,   8.,   9.,  12.,  16.],
                [  1.,   5.,   6.,   7.,  25.,  30.,  35.,  36.,  42.,  49.]])
```

By default mit *Bias-Term*, also vorangehender `1`.

Aber **Achtung**: das Hinzufügen zahlreicher 'künstlich erzeugter' *Features* erhöht die Anzahl Freiheitsgrade und kann sehr leicht zu *Overfitting* führen. (Siehe weiter unten.) Es empfiehlt sich, bloss bei begründetem Verdacht auf nicht-lineare Zusammenhänge zusätzliche nicht-lineare *Features* zu erzeugen.

Standardisieren

Eine weitere Methode des Präprozessierens ist das Standardisieren:

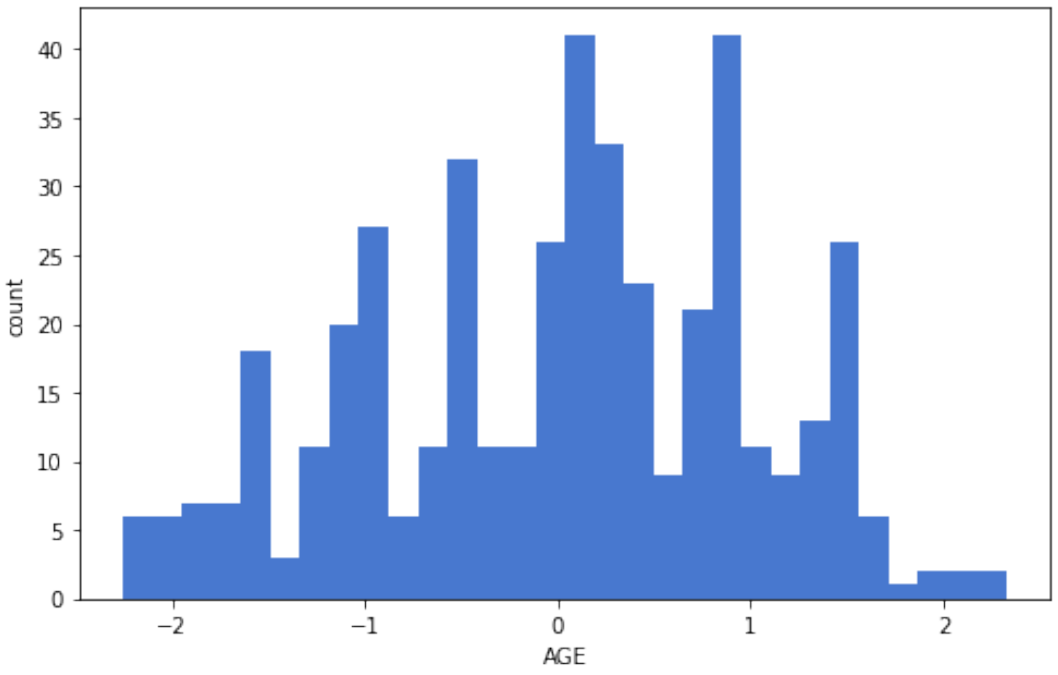
Oftmals werden Datensätze standardisiert, das heisst so transformiert, dass die Variablen **Mittelwert gleich 0** (*zentrieren*) und **Standardabweichung/Varianz gleich 1** haben.

Es gibt verschiedene Gründe warum dies sinnvoll sein kann: unter anderem weil zahlreiche **Algorithmen**, welche zum Lernen von Machine Learning Modellen verwendet werden / zum Finden der optimalen Modell-Koeffizienten **stabiler**, bzw. oft auch **schneller** sind mit standardisierten Daten.

Ein weiterer Grund liegt in der **Wirkung der Regularisierung**, wie wir etwas weiter unten sehen werden.

```
In [40]: Xstd = (X - X.mean(axis=0))/X.std(axis=0)
         ystd = (y - y.mean())/y.std()
```

```
In [41]: # AGE
         fig, ax = plt.subplots(figsize=(8, 5))
         # plot into axes
         _ = ax.hist(Xstd[:,0], bins=30)
         # label axes
         _ = ax.set_xlabel('AGE')
         _ = ax.set_ylabel('count')
```



```
In [42]: lr = LinearRegression(fit_intercept=True).fit(Xstd, ystd)

         lr.score(Xstd, ystd)
```

```
Out[42]: 0.51774842222034989
```

```
In [43]: lr.intercept_
```

```
Out[43]: -6.2123864735367238e-16
```

```
In [44]: lr.coef_
```

```
Out[44]: array([-0.00618293,  0.32110005,  0.20036692, -0.48931352,  0.29447365,
                 0.06241272,  0.10936897,  0.46404908,  0.04177187,  0.14813008])
```

Neben der Standardisierung gibt es noch eine Reihe weiterer Möglichkeiten Variablen zu skalieren und zu transformieren. Einige weitere davon sind in `scikit-learn` hier beschrieben und implementiert. Wir betrachten sie aber vorerst nicht weiter.

3 Bias-Variance Trade-Off

Wollen wir Prognosen machen, so interessiert uns nicht nur wie gut ein Modell Datenpunkte abschätzt die es schon kennt, also solche, welche zum 'Trainieren' verwendet wurden, sondern vor allem wie gut es den Output von unbekannten Datenpunkten vorherzusagen vermag.

Systematische Ansätze, das beste Modell für die Vorhersage unbekannter Datenpunkte zu finden werden wir gegen Ende des Semsters untersuchen. Hier betrachent wir aber nun bereits drei Konzepte aus diesem Themengebiet:

Over- und Underfitting

Bias vs. Variance

Betrachtet man ein allgemeines Regressionsproblem, so gilt es eine Funktion $f(x, \hat{\beta})$ zu finden mit

$$\hat{\beta} = \text{argmin}_{\beta} \quad J(y^{(train)}, f(x^{(train)}, \beta), \beta) \tag{9}$$

Dabei lässt sich zeigen, dass sich der Erwartungswert des quadrierten Fehlers für ein gegebenes Regressionsmodell $f(x, \hat{\beta})$ und Datenpunkten $(x^{(test)}, y^{(test)})$ ausserhalb des Trainings-Sets wie folgt zerlegen lässt:

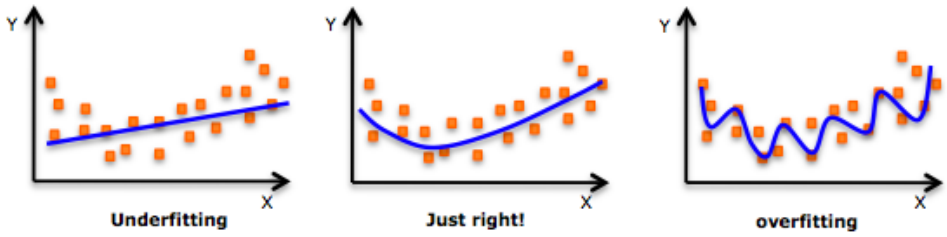


Illustration over- / under-fitting

Illustration Bias-Variance Trade-off

$$E\left[(y^{(test)} - f(x^{(test)}, \hat{\beta}))^2\right] = \left(E\left[f(x^{(test)}, \hat{\beta})\right] - f(x^{(test)}, \beta_{true})\right)^2 + E\left[\left(f(x^{(test)}, \hat{\beta}) - E[f(x^{(test)}, \hat{\beta})]\right)^2\right] + \sigma^2 \tag{10}$$

Hierbei ist auf der rechten Seite der erste Term der **quadrierte Bias**, der zweite Term die **Varianz** und σ^2 der irreduzierbare Fehler.

Also:
Erwartungswert des quadrierten Fehlers = **quadrierter BIAS** + **VARIANZ** + *irreduzierbarer Fehler*

Wie gross der **Bias** und wie gross die **Variance** ausfällt ist, bei gegebener Funktion $f(\cdot)$, eine Frage der Wahl der Modell-Koeffizienten β.

Im Falle zahlreicher korrelierter Input-Variablen kann *Overfitting* dadurch begegnet werden, dass man die Grösse der Koeffizienten beschränkt. Dies kann man durch **Regularisierung** erreichen.

Das Ziel dabei ist, dem Modell **Bias** zuzuführen, aber dafür **Variance** zu nehmen und damit *Overfitting* zu vermeiden. Dies kann für die Vorhersage neuer Werte einen positiven Effekt haben.

'*Trainings*'- und '*Test*'-Daten

Dem Problem, dass man die Stärke eines Modells auf Datenpunkten testen möchte, welche vom Algorithmus nicht 'gesehen' wurden während des Trainings, kann einfach begegnet werden indem wir unseren Datensatz unterteilen in '*Trainings*'- und '*Test*'-Daten. Das heisst, vom Datensatz welcher uns vorliegt und *Input* und *Output* beinhaltet, entnehmen wir einen Teil, welchen wir im Anschluss zum 'Training' / Fit zum 'Testen' der Stärke unseres Modells brauchen werden.

Ein oft verwendetes Verhältnis ist 80 % : 20 % von Trainings- zu Test-Daten.

Die Unterteilung in Trainings- und Test-Daten muss dabei **randomisiert** erfolgen, um zu vermeiden, dass in der Riehenfolge der Daten ein Faktor liegt, welcher das Resultat beeinflussen könnte.

```
In [45]: # the following code can be used to pick points in random order
         NUM_POINTS = 10
         np.random.permutation(range(NUM_POINTS))

         # .. think about how to pick in 80 : 20 proportion ..
```

```
Out[45]: array([6, 5, 0, 9, 2, 4, 1, 8, 3, 7])
```

Das Resultat ist schlussendlich der berechnete Wert eines Metrik auf den Test-Daten. Im Falle der Regression ist das **Bestimmtheitsmass** eine gute Metrik, um die Stärke des Modells zu bemessen.

4 Regularisierung

Ridge Regression

$$J(\beta) = \sum_{i=1}^{N}(y^{(i)} - x^{(i)}\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \tag{11}$$

Kann mit **Gradient Descent** minimiert werden:

$$\big(\nabla J(\beta)\big)_j = \frac{\partial J(\beta)}{\partial \beta_j} = -2\big(\sum_{i=1}^{N}(y^{(i)} - x^{(i)} \cdot \beta)x_j^{(i)}\big) + 2\lambda\beta_j \tag{12}$$

Normalengleichung

$$\hat{\beta} = (X^T X + \lambda \mathbf{1})^{-1} X^T y \tag{13}$$

- $\mathbf{1}$ ist eine Diagonalmatrix der Dimension $(p \times p)$ mit 1-en auf der Diagonalen und 0-en sonst.

- die Matrix $(X^T X + \lambda \mathbf{1})$ ist für $\lambda > 0$ **immer** invertierbar.
- damit sämtliche Koeffizienten gleich skaliert sind, muss der **Datensatz standardisiert** werden.
- entspricht linearer Regression mit Nebenbedingung $\sum_{i=1}^{p} \beta_i^2 \leq t$

```
In [46]: from sklearn.linear_model import Ridge
```

```
In [47]: rr = Ridge(fit_intercept=False, alpha=1.)
```

```
In [48]: rr.fit(Xstd, ystd)
```

```
Out[48]: Ridge(alpha=1.0, copy_X=True, fit_intercept=False, max_iter=None,
             normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
In [49]: rr.score(Xstd, ystd)
```

```
Out[49]: 0.51758216340630403
```

Vergleich Ridge-Regression-Regularisierung-Pfade mit / ohne Standardisierung

---

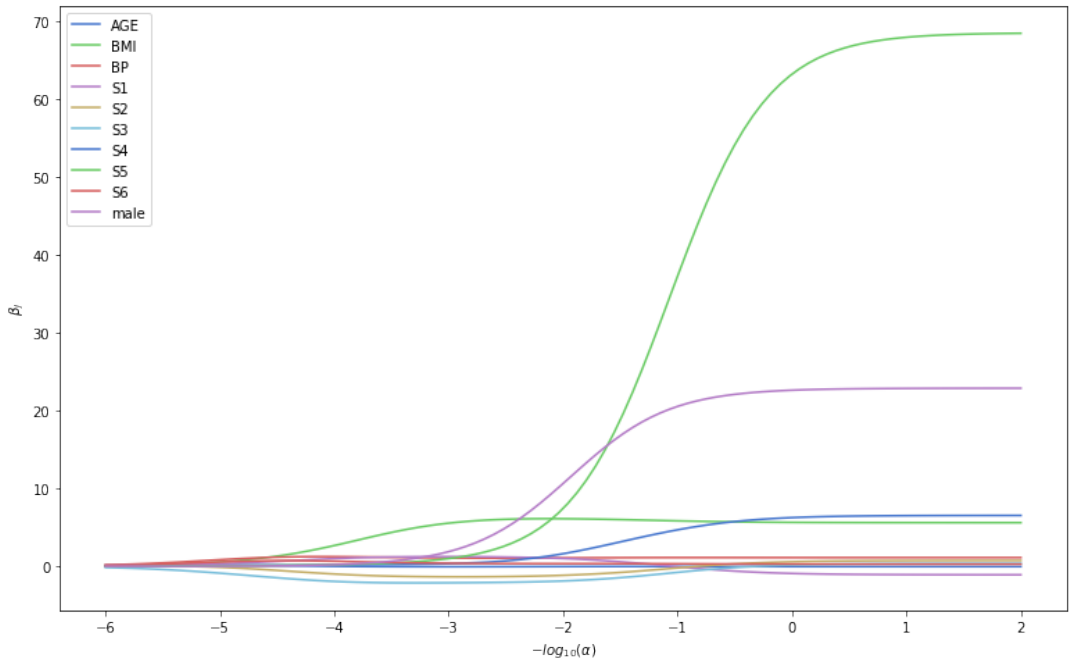
```
In [50]: Ridge(fit_intercept=True, normalize=True, alpha=1.).fit(X, y).score(X,y)
```

```
Out[50]: 0.45123062774361744
```

```
In [51]: alphas = np.logspace(-2, 6, 100)
         coefs_ = np.array([Ridge(fit_intercept=True, alpha=ai).fit(X, y).coef_ for ai in
         alphas])
```

```
In [52]: predictors = ['AGE', 'BMI', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5',
         'S6','male','female',]
         fig, ax = plt.subplots(figsize=(13, 8))
         for i in range(coefs_.shape[1]):
             _ = ax.plot(-np.log10(alphas), coefs_.T[i], label=predictors[i])

         _ = ax.set_ylabel(r'$\beta_j$')
         _ = ax.set_xlabel(r'$-log_{10}(\alpha)$')
         _ = ax.legend()
```

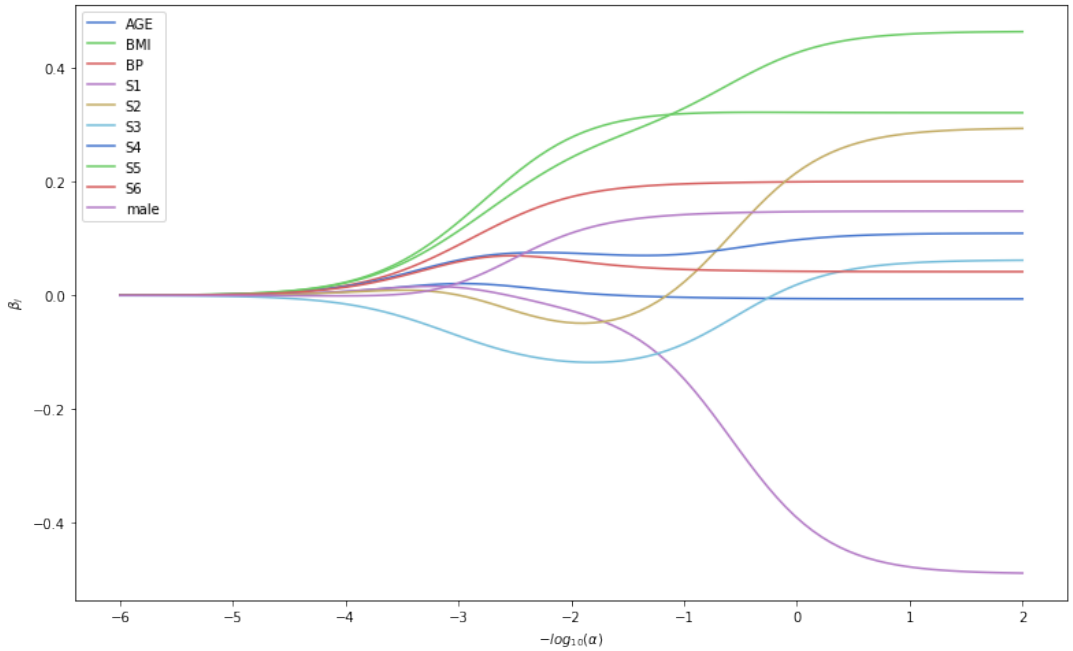


```
In [53]: alphas = np.logspace(-2, 6, 100)
         coefs_ = np.array([Ridge(fit_intercept=False, alpha=ai).fit(Xstd, ystd).coef_
         for ai in alphas])
```

```
In [54]: predictors = ['AGE', 'BMI', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5',
         'S6','male','female',]
         fig, ax = plt.subplots(figsize=(13, 8))
```

```
for i in range(coefs_.shape[1]):
    #_ = ax.plot(-np.log10(alphas), (coefs_*X.std(axis=0)).T[i],
label=predictors[i])
    _ = ax.plot(-np.log10(alphas), coefs_.T[i], label=predictors[i])


_ = ax.set_ylabel(r'$\beta_j$')
_ = ax.set_xlabel(r'$-log_{10}(\alpha)$')
_ = ax.legend()
```



Lasso

Verwenden wir anstelle der Summe der quadrierten Koeffizienten β_j die Summe der Beträge als 'Strafterm', so erhalten wir folgende *Cost-Function*:

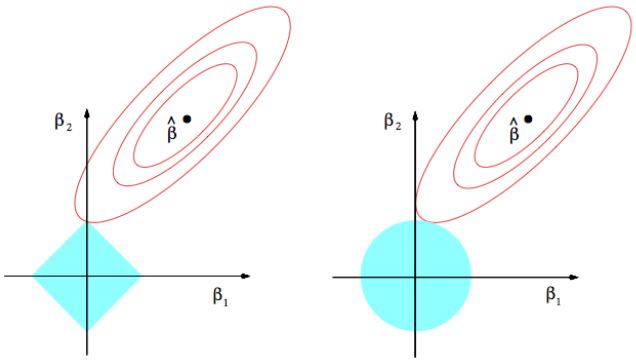
$$J(\beta) = \sum_{i=1}^{N} (y^{(i)} - x^{(i)}\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{14}$$

Oftmals wird dieser Ansatz als **Lasso** bezeichnet.

Die Lasso *Cost-Function* kann nicht mehr mit *Gradient Descent* minimiert werden, da der Gradient des zweiten Terms nicht stetig ist. Anstelle dessen kann **Coordinate Descent** verwendet werden ..

Das Lasso hat intrinsische *feature selection*-Eigenschaft

```
In [55]: from sklearn.linear_model import Lasso, lasso_path
```



Lasso and Ridge Regression in the Coefficient Plane

```
In [56]: lasso = Lasso(fit_intercept=False, alpha=.1)
         lasso.fit(Xstd, ystd)
```

```
Out[56]: Lasso(alpha=0.1, copy_X=True, fit_intercept=False, max_iter=1000,
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False)
```

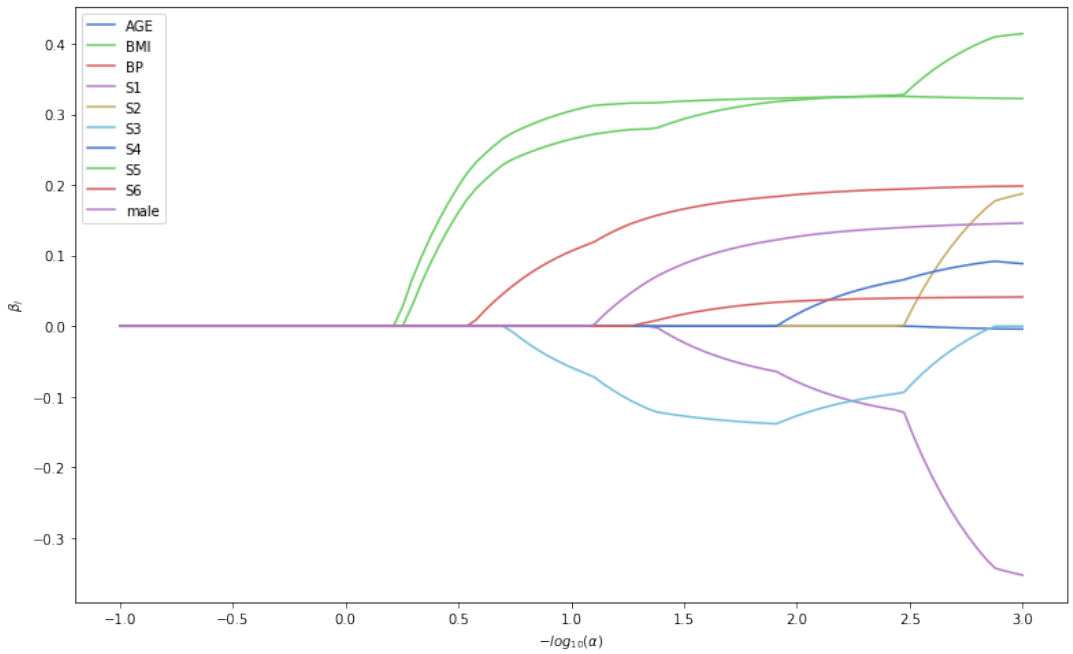
```
In [57]: lasso.coef_
```

```
Out[57]: array([ 0.        ,  0.30487704,  0.10631091, -0.        , -0.        ,
                -0.05842263,  0.        ,  0.26474254,  0.        ,  0.        ])
```

Der Lasso-Koeffizienten-Pfad

```
In [58]: alphas_lasso, coefs_lasso, _ = lasso_path(Xstd, ystd, alphas=np.logspace(-3, 1,
         100))
```

```
In [59]: predictors = ['AGE', 'BMI', 'BP', 'S1', 'S2', 'S3', 'S4', 'S5',
         'S6','male','female',]
         fig, ax = plt.subplots(figsize=(13, 8))
         for i in range(coefs_lasso[1].shape[0]):
             try:
                 _ = ax.plot(-np.log10(alphas_lasso), coefs_lasso[i],
         label=predictors[i])
             except:
                 pass
         _ = ax.set_ylabel(r'$\beta_j$')
         _ = ax.set_xlabel(r'$-log_{10}(\alpha)$')
         _ = ax.legend()
```



Coordinate Descent

Coordinate Descent ist genauso wie *Gradient Descent* ein *Descent Algorithmus*, allerdings schreitet man nicht in Richtung des negativen Gradienten dem Minimum der *Cost-Function* entgegen. Stattdessen *'loopt'* man wiederkehrend durch die verschiedenen Koeffizienten und aktualisiert nur den jeweils aktuellen (i.e. die aktuelle Koordinate). Daher rührt der Name *Coordinate Descent*.

In der Regel initialisiert man die Koeffizienten mit der *Least Squares*-Lösung.

Wenn sie eine Herleitung des Algorithmus einsehen möchten, so schauen sie sich *Friedman et al., Pathwise Coordinate Optimization, 2007* an, oder *Hastie et al., Elements of Statistical Learning, 2013, pp 92-93*.