Fachhochschule Nordwestschweiz
Hochschule für Technik

Machine Learning

FS2019

Lerneinheit 2

Lineare Regression, Gradient Descent, Normalengleichung

Dozent: Michael Graber
michael.graber@fhnw.ch

Diese Vorlesungsunterlagen sind aus den Folien der Vorlesungen zusammengestellt worden.

Organisatorisches

Mattermost Team

- Eine Einladung um unserem 'Team' beizutreten, sollten Sie per Email erhalten haben.
- Hier werden wir Fragen zu den Übungen, Ankündigungen etc für alle einfach sichtbar miteinander diskutieren.

Abgabetermin der nicht bewerteten Übungen

- jeweils Mittwoch Abend 24 Uhr, vor der nächsten Vorlesung.
- via *Submit* auf `mlhub`.

Lernziele

- Sie wissen was *einfache* und *multiple lineare Regression* ist und kennen den Zusammenhang mit einem statistischen *linearen Modell*.
- Sie kennen die Ausgangslage in welcher *lineare Regression* angewendet werden kann.
- Sie kennen *Gradient Descent* und die Normalengleichung als Lösungsansätze für den *Least Squares*-Ansatz zum Auffinden der Modell-Koeffizienten und wissen wann diese eingesetzt werden können / sollen.

- Sie wissen wie sie den Anteil der Varianz bestimmen können, welcher von einem Modell erklärt wird und wissen was diese Grösse (das Bestimmtheitsmass) aussagt.
- Sie kennen den *Tukey-Ascombe Plot* und können ihn zur Interpretation einer linearen Regression einsetzen.
- Sie kennen die grundlegende API von `scikit-learn` und können diese im Falle der linearen Regression verwenden.

1 Ein lineares Modell

Ein lineares statistisches Modell

$$y^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \cdots + \beta_N x_N^{(i)} + \varepsilon^{(i)} \tag{1}$$

beschreibt einen linearen Zusammenhang zwischen *Input*- und *Output*-Werten eines Datensatzes $(\mathbf{X}, y)$. Das Modell beinhaltet einen 'Achsenabschnitt' β_0, Koeffizienten β_j für die *Input*-Variablen und einen unbekannten, 'zufälligen' Fehler $\varepsilon^{(i)}$. Wir verlangen, dass die Fehler $\varepsilon^{(i)}$ unabhängig voneinander alle aus der gleichen Normalverteilung stammen:

$$\varepsilon^{(i)} \sim \mathcal{N}(0, \sigma) \tag{2}$$

Ergänzen wir den *Input*-Vektor $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \cdots, x_N^{(i)})$ um eine 1 am Anfang $x^{(i)} = (1, x_1^{(i)}, x_2^{(i)}, \cdots, x_N^{(i)})$, so können wir Gleichung 1 in Vektorschreibweise wie folgt schreiben:

$$y^{(i)} = x^{(i)}\beta + \varepsilon^{(i)} \tag{3}$$

mit $\beta = (\beta_0, \beta_1, \cdots, \beta_P)$.
Für den ganzen Datensatz können wir nun das folgende Modell formulieren:

$$y = \mathbf{X}\beta + \varepsilon \tag{4}$$

mit $\varepsilon = (\varepsilon^{(1)}, \varepsilon^{(2)}, \cdots, \varepsilon^{(N)})$.

Annahmen für das lineare Modell

- Erwartungswert der Fehlervariablen $E[\varepsilon_i] = 0$
- Die Fehlervariablen sind unkorreliert : $Cov(\varepsilon_i, \varepsilon_j) = 0$
- Die Varianz der Fehlervariablen ist konstant : $Var(\varepsilon_i) = \sigma$
- 'Messungen' $x^{(i)}$ sind exakt

2 Lineare Regression

Lineare Regression ist das Verfahren jene Koeffizienten β eines linearen Modells zu finden, welche den Datensatz $(\mathbf{X}, y)$ 'bestmöglich' beschreiben.

Solange die Koeffizienten β nicht fixiert sind, können wir den *Output* y in Gleichung 3 allgemein als Funktion sowohl von x wie von β betrachten. Bei Vernachlässigung des Fehlers ε, auf welchen wir keinen Einfluss haben beim Suchen des 'besten' Modells, können wir dann das folgende **hypothetische lineare Modell** formulieren:

$$l(\beta, x^{(i)}) = x^{(i)}\beta = \beta_0 \cdot 1 + \beta_1 x_1^{(i)} + \cdots + \beta_N x_N^{(i)} \tag{5}$$

Unter Zuhilfenahme dieses hypothetischen Modells $l(\beta, x)$ können wir für unseren Datensatz $(\mathbf{X}, y)$ eine Funktion für den Fehler definieren. Die am häufigsten verwendete **Fehlerfunktion** für Regressionsprobleme ist die Summe der Fehlerquadrate:

$$J(\beta) = \sum_{i=1}^{N} (y^{(i)} - l(\beta, x^{(i)}))^2 \tag{6}$$

Die Differenz einer hypothetischen Funktion und dem eigentlichen Messwert für einen Datenpunkt bezeichnet man oft als **Residuum**, woher der häufig verwendete englische Ausdruck ***residual sum of squares*** stammt.

Was wir nun vor uns haben ist ein Optimierungsproblem. Es gilt ein Minimum der Fehlerfunktion $J(\beta)$ zu finden. Bei Optimierungsproblemen spricht man oft von der ***cost function*** oder der ***objective function*** anstelle der Fehlerfunktion.

Einfache lineare Regression

Betrachten wir vorerst einen einfachen Fall: Wir haben bloss eine 1-dimensionale *Input*-Variable X und eine 1-dimensionale *Output*-Variable Y. Das dazugehörige lineare Modell schaut also wie folgt aus:

$$y^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \varepsilon^{(i)} \tag{7}$$

Unsere Fehlerfunktion wird nun zu:

$$J(\beta_0, \beta_1) = \sum_{i=1}^{N} (y^{(i)} - l(\beta_0, \beta_1, x^{(i)}))^2 = \sum_{i=1}^{N} (y^{(i)} - (\beta_0 + \beta_1 x_1^{(i)}))^2 \tag{8}$$

Eine Illustration dazu finden Sie im folgenden Bild. Um die optimale Regressionsgerade (dunkelblau) zu finden, gilt es, die Summe der senkrechten Abstände (hellgrau) der Datenpunkte zur Geraden zu minimieren. Die alternativen hypothetischen linearen Funktionen (hellblau) haben allesamt eine grössere Summe quadrierter Fehler.

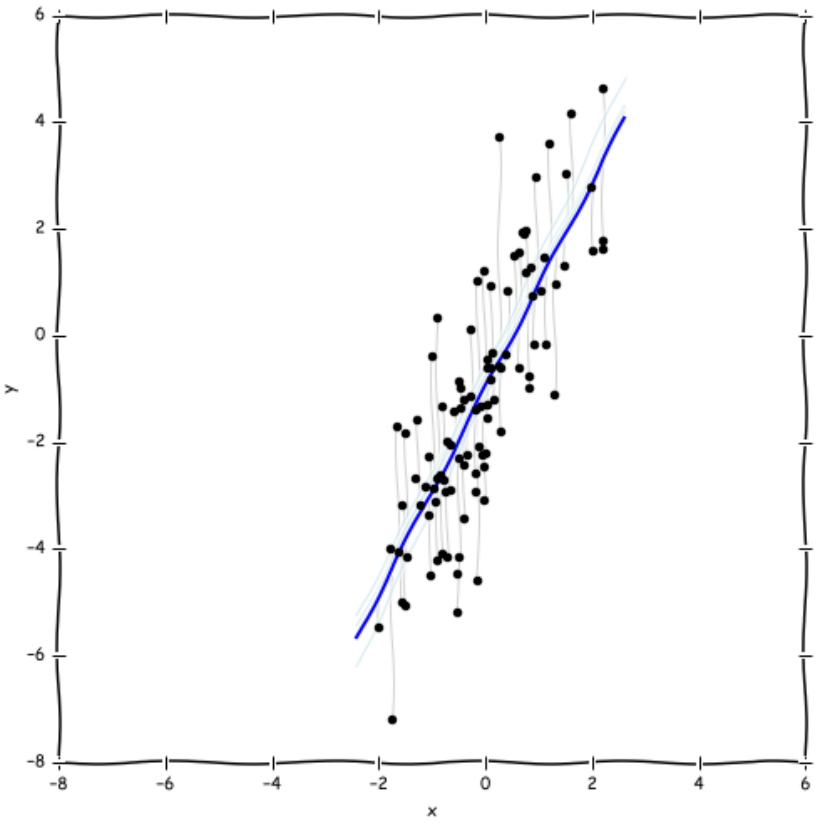
Methode der kleinsten Fehlerquadrate (*Least Squares*)

Die Form der *Least Squares cost function*

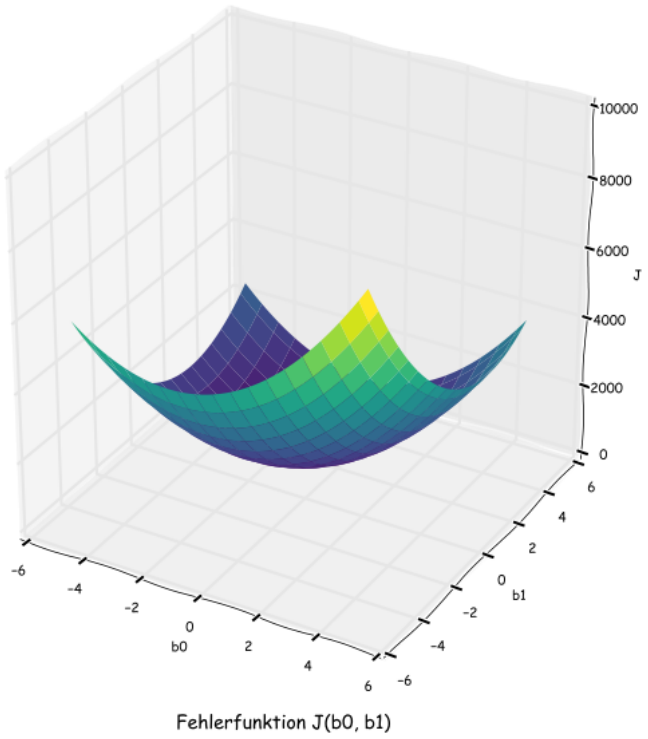
Das **Minimum oder Maximum einer differenzierbaren Funktion zu finden**, ist ein Problem, welches Sie bereits aus dem Analysis-Unterricht kennen. Wir betrachten hier dafür zwei Lösungsansätze:

1. Wir können **analytisch** die Ableitung (den Gradienten) der Funktion bilden und diese(n) gleich Null setzen. Damit finden wir jene Punkte, an welchen keine Steigung vorhanden ist, jene also, welche Kandidaten für Extremalstellen sind.
2. Wir können mit einem beliebigen Startwert für die Koeffizienten β beginnen und uns dann **iterativ** 'in günstiger Richtung' hin zu kleineren Funktionswerten bewegen.

Den ersten Ansatz werden wir im Kontext mehrerer Koeffizienten, der **multiplen linearen Regression** betrachten und uns hier vorerst dem Gradientenverfahren widmen:



Methode der kleinsten Fehlerquadrate



Oberfläche einer *Least Squares Cost Function*.

Gradient Descent

Der Gradient einer Funktion

Betrachtet man eine Funkion $f(x) : \mathbf{R}^n \to \mathbf{R}$ so ist der Gradient ∇f der Vektor der partiellen Ableitungen nach sämtlichen Variablen x_i:

$$\nabla f(x)_i = \frac{\partial f(x)}{\partial x_i} \tag{9}$$

Der Gradient ist somit 'die mehrdimensionale Version' der einfachen Ableitung. Da er die Steigungen in sämtliche Richtungen kombiniert, zeigt er in die Richtung des grössten Zuwachses der Funktionswerte von $f(x)$ für beliebige Punkte x. Natürlich ist er bloss definiert für den Fall, dass f nach den Variablen x_i auch *differenzierbar* ist.

Gradient Descent gehört zur grösseren Menge der *Descent Methoden*. Ihnen allen gemein ist der iterative Ansatz, wie auch der Umstand dass die Richtung in welche wir uns iterativ bewegen einen spitzen Winkel mit dem *negativen Gradienten* der Funktion einnehmen muss. Es muss aber nicht zwingend der Gradient selber sein, welcher die Richtung hin zu kleineren Funktionswerten vorgibt. Zudem müssen wir für alle *Descent Methoden* auch die *Schrittlänge* wählen.

Der ***Gradient Descent*-Algorithmus** funktioniert wie folgt:

Initialisiere $x^{(0)}$.

nicht k (hoch)?

Aktualisiere $x^{(k)} = x^{(k-1)} - \eta \cdot \nabla f(x^{(k-1)})$

bis

Wiederholde den Aktualisierungsschritt für eine bestimmte Anzahl Schritte, die Länge des Gradienten im Punkt $x^{(k)}$ sehr klein ist ($\|\nabla f(x^{(k)})\|_2 < \epsilon$), oder ein anderes Kriterium erfüllt ist.

(Hier bezeichnet die hochgestellte Zahl i in Klammern den i-ten Iterationsschritt.)

Äquipotentiallinien der *Least Squares Cost Function*

Der Gradient der Fehlerfunktion $J(\beta)$

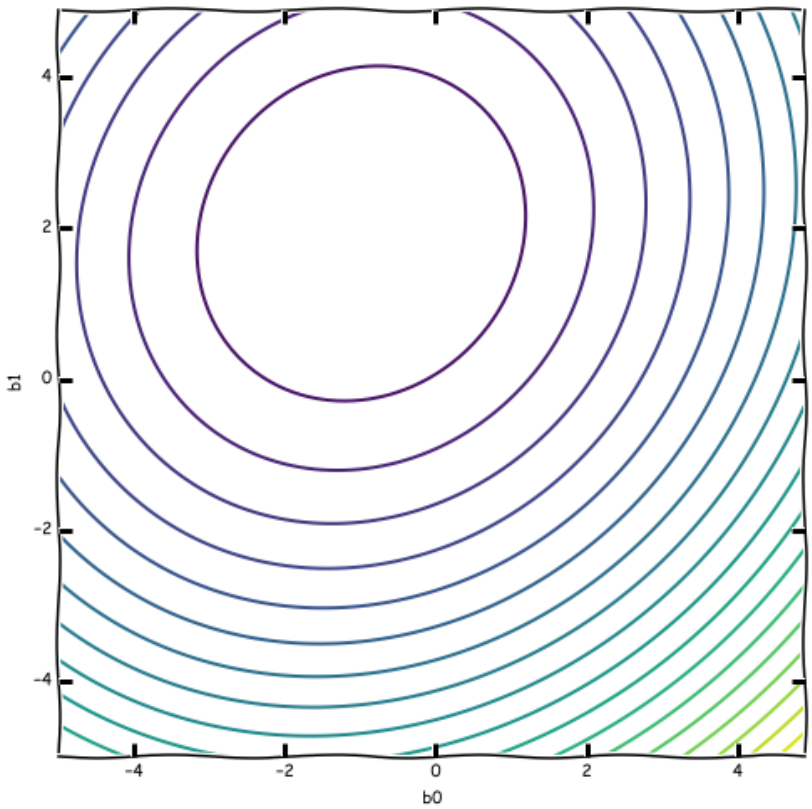
Die Funktion, welche uns im Zusammenhang mit linearer Regression interessiert, ist die Fehlerfunktion $J(\beta)$. Für sie wollen wir ein Minimum finden. Also gilt es für's Gradientenverfahren ihren Gradienten zu bestimmen:

$$\nabla J(\beta) = \left(\frac{\partial J(\beta)}{\partial \beta_0}, \frac{\partial J(\beta)}{\partial \beta_2}, \ldots, \frac{\partial J(\beta)}{\partial \beta_p}\right) \tag{10}$$

Und dabei gilt:

$$\frac{\partial J(\beta)}{\partial \beta_j} = \frac{\partial}{\partial \beta_j} \sum_{i=1}^{N} (y^{(i)} - \overbrace{l(\beta, x^{(i)})}^{\text{Funktion l(B,x)}})^2 = \frac{\partial}{\partial \beta_j} \sum_{i=1}^{N} (y^{(i)} - \overbrace{x^{(i)} \cdot \beta}^{\text{Skalarprodukt}})^2 = \sum_{i=1}^{N} \overbrace{\frac{\partial}{\partial \beta_j}}^{\text{Innere Ableitung (summe ignorieren)}} (y^{(i)} - x^{(i)} \cdot \beta)^2 = \overbrace{-2 \sum_{i=1}^{N} (y^{(i)} - x^{(i)} \cdot \beta) x_j^{(i)}}^{\text{abgeleitet}} \tag{11}$$

- (Oftmals wird der Fehlerfunktion $J(\beta)$ noch ein Faktor $\frac{1}{2}$ vorgestellt, damit der Faktor 2 aus dem Gradienten verschwindet. Das Finden der Minimalstelle der Funktion hat dasselbe Resultat.)



Contour Plot der *Cost Function* einer linearen Regression mit zwei Koeffizienten.

- Der hergeleitete Gradient $\nabla J(\beta)$ ist bereits für p *Input*-Variablen formuliert.

Nun haben wir für eine erste Implementation sämtliche Zutaten beisammen:

Die `scikit-learn` API

- 'Estimator' / 'Predictor' Klassen
- Initialisierung definiert Modell
- `.fit(X, y)` - Methode
- `.predict(X)` - Methode
- `.coef_` und `intercept_` - Attribut

```
In [1]: from sklearn.linear_model import LinearRegression as skLinearRegression
```

```
In [2]: print(skLinearRegression.__doc__)
```

```
    Ordinary least squares Linear Regression.

    Parameters
    ----------
    fit_intercept : boolean, optional, default True
        whether to calculate the intercept for this model. If set
        to False, no intercept will be used in calculations
        (e.g. data is expected to be already centered).

    normalize : boolean, optional, default False
        This parameter is ignored when ``fit_intercept`` is set to False.
        If True, the regressors X will be normalized before regression by
        subtracting the mean and dividing by the l2-norm.
        If you wish to standardize, please use
        :class:`sklearn.preprocessing.StandardScaler` before calling ``fit`` on
        an estimator with ``normalize=False``.

    copy_X : boolean, optional, default True
        If True, X will be copied; else, it may be overwritten.

    n_jobs : int, optional, default 1
        The number of jobs to use for the computation.
        If -1 all CPUs are used. This will only provide speedup for
        n_targets > 1 and sufficient large problems.

    Attributes
    ----------
    coef_ : array, shape (n_features, ) or (n_targets, n_features)
        Estimated coefficients for the linear regression problem.
```

```
        If multiple targets are passed during the fit (y 2D), this
        is a 2D array of shape (n_targets, n_features), while if only
        one target is passed, this is a 1D array of length n_features.

    intercept_ : array
        Independent term in the linear model.

    Notes
    -----
    From the implementation point of view, this is just plain Ordinary
    Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.
```

```
In [3]: sklr = skLinearRegression()
        print(sklr.fit.__doc__)


        Fit linear model.

        Parameters
        ----------
        X : numpy array or sparse matrix of shape [n_samples,n_features]
            Training data

        y : numpy array of shape [n_samples, n_targets]
            Target values. Will be cast to X's dtype if necessary

        sample_weight : numpy array of shape [n_samples]
            Individual weights for each sample

            .. versionadded:: 0.17
               parameter *sample_weight* support to LinearRegression.

        Returns
        -------
        self : returns an instance of self.
```

.. wir implementieren die Lineare Regression nun aber selbst !

```
In [4]: import numpy as np

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

In [5]:
```
class LinearRegressionGradientDescent(object):
    '''
    Implements a Gradient Descent Optimization for a Linear Regression Problem.
    '''

    def __init__(self, init_coef=(0., 0.), epsilon=0.00001, maxsteps=1000,
                 stepsize=0.001, linesearch='fixed'):
        '''
        Gradient Descent Optimizer for Linear Regression.
        '''
        self.init_coef = init_coef
        self.epsilon = epsilon
        self.maxsteps = maxsteps
        self.linesearch = linesearch
        self.stepsize = stepsize

        self.coef_ = np.array(self.init_coef)
        self._nsteps = 0

        print('LinearRegressionGradientDescent(init_coef={}, epsilon={},\n
maxsteps={},'.format(
                        self.init_coef, self.epsilon, self.maxsteps) +
                    ' linesearch={}, stepsize={})\n'.format(self.linesearch,
self.stepsize))


    def fit(self, X, y, verbose=False):
        '''
        Fits the coefficients beta of a linear regression problem
        to the dataset (X, y).
        '''

        # keept track of the the norm/length of the gradient
        normgrad = []
        neg_grad = -self.least_squares_gradient(self.coef_, X, y)
        normgrad.append(np.linalg.norm(neg_grad))

        while not self._do_stop(X, y):

            if self.linesearch == 'fixed':
                self.coef_ += self.stepsize*neg_grad

            elif self.linesearch == 'backtracking':
                stepsize = self.backtracking_linesearch(X, y)
                self.coef_ += stepsize*neg_grad

            # calculate the the norm of the gradient
            neg_grad = -self.least_squares_gradient(self.coef_, X, y)
```

```
            normgrad.append(np.linalg.norm(neg_grad))

            if verbose:
                print('step {s} : neg grad = {ng}, normgrad: {nog},  coef_ =
{c}, stepsize = {ss}'.format(
                        s=self._nsteps, ng=neg_grad, c=self.coef_,
                        nog=np.linalg.norm(neg_grad),
                        ss=self.stepsize if self.linesearch == 'fixed' else
stepsize ))

            self._nsteps += 1

        self.normgrad_ = np.array(normgrad)

        print('Done fitting! \n')

        print('n steps : ', self._nsteps)
        print('norm gradient : ', self.normgrad_[-1])

        return self


    def _do_stop(self, X, y):
        grad = self.least_squares_gradient(self.coef_, X, y)
        tiny = np.linalg.norm(grad) < self.epsilon
        manysteps = self._nsteps >= self.maxsteps

        return (tiny or manysteps)


    @staticmethod
    def least_squares_gradient(coef, X, y):
        '''
        Calculates the least squares gradient at position coef
        for the dataset (X, y).
        '''
        grad = []
        for idx in range(X.shape[1]):
            grad.append(((y - coef.dot(X.T))*X[:, idx]).sum())
        return -2*np.array(grad)


    def predict(self, X):
        return X.dot(self.coef_)
```

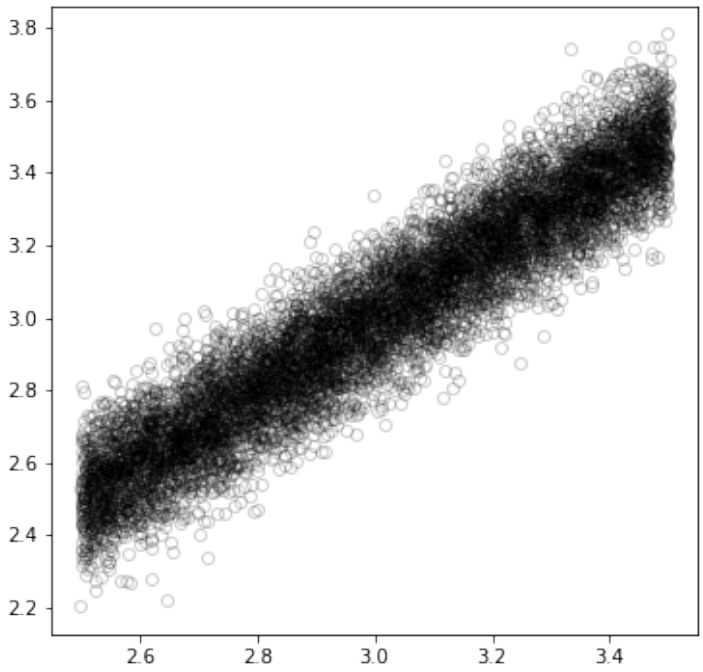
Und nun erzeugen wir uns einen Datensatz an welchem wir unsere Implementation der linearen Regression testen können:

```
In [6]: def generate_random_linear_data(n_points=100, b0=0., b1=1., sigma=1., mux=3.):
```

```
    '''
    Generates uniformiformly distributed x with mean mux and
    y according to a linear model.
    '''

    x = np.random.rand(n_points)+mux-0.5
    y = b0 + x*b1 + sigma*np.random.randn(n_points)

    return x, y
```

In [7]:
```
NPOINTS = 10000

X, y  = generate_random_linear_data(n_points=NPOINTS, sigma=0.1)

X = np.array([np.ones(NPOINTS), X]).T
```

In [8]:
```
fig, ax = plt.subplots(figsize=(6,6))
_ = ax.plot(X[:,1], y, 'o', markeredgecolor='black', color='none', alpha=0.2)
```



In [9]:
```
lr = LinearRegressionGradientDescent(init_coef=(-1., -1.), stepsize=0.000001,
epsilon=.1**3,
                                     maxsteps=100000)

_ = lr.fit(X, y)
```

```
LinearRegressionGradientDescent(init_coef=(-1.0, -1.0), epsilon=0.0010000000000000002,
 maxsteps=100000, linesearch=fixed, stepsize=1e-06)

Done fitting!

n steps :  65906
norm gradient :  0.000999888199268
```

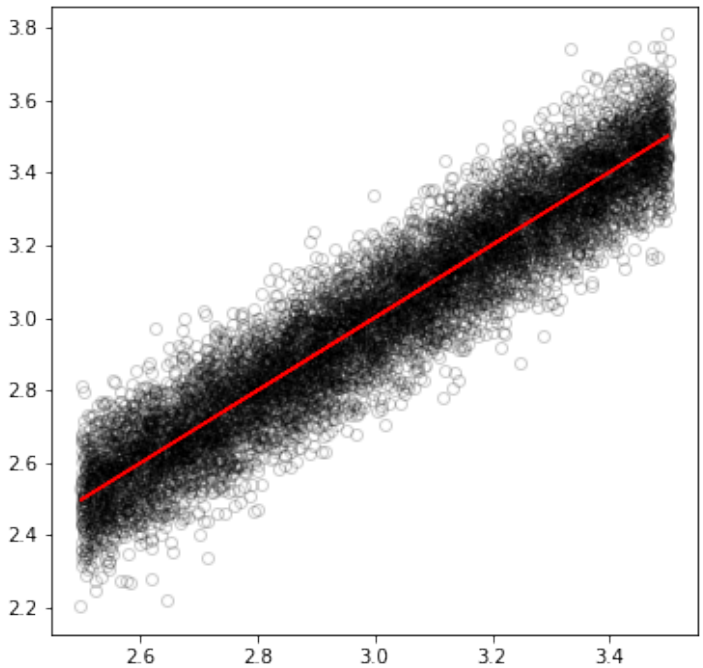
In [10]:
```
lr.coef_
```

Out[10]:
```
array([-0.01759491,  1.00570417])
```

In [11]:
```
lr._nsteps
```

Out[11]:
```
65906
```

In [12]:
```
b, a = lr.coef_
fig, ax = plt.subplots(figsize=(6,6))
_ = ax.plot(X[:,1], y, 'o', markeredgecolor='black', color='none', alpha=0.2)
_ = ax.plot(X[:, 1], a*X[:, 1]+b, '-r')
```



Berechnung der Residuen

$$r^{(i)} = y^{(i)} - x^{(i)}\beta \tag{12}$$

```
In [13]: predicted = lr.predict(X)
         residuals = y - predicted
```

Das Bestimmtheitsmass R^2

Wenn wir ein Mass definieren möchten, welches beschreibt, wie gut unser Modell den Datensatz beschreibt, so eignet sich das Bestimmtheitsmass R^2.

Das Bestimmtheitsmass berechnet wieviel der gesamten Varianz der abhängigen Variablen y durch das Modell und die Koeffizienten β erklärt werden kann:

$$R^2 = 1 - \frac{Var(r)}{Var(y)} \tag{13}$$

```
In [20]: 1. - ((residuals - residuals.mean())**2).sum()/((y-y.mean())**2).sum()
```

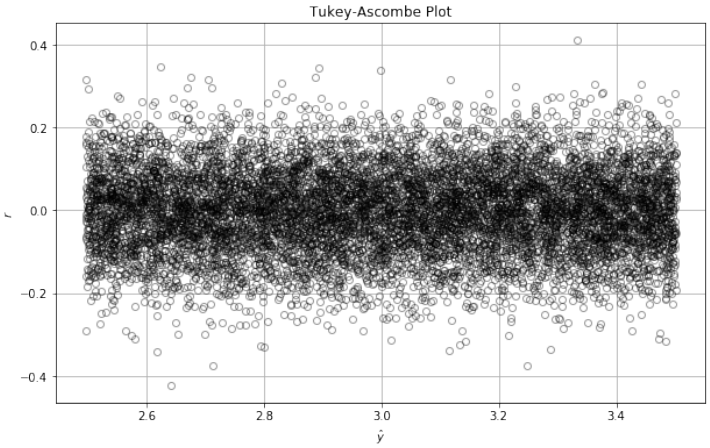
```
Out[20]: 0.89254799055982836
```

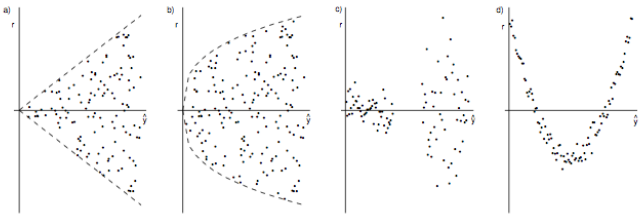
Tukey - Ascombe Plot

Verifikation der Modellannahmen!

Wir plotten die Residuen r als Funktion der geschätzten Werte $\hat{y}$ für sämtliche *Input*-Punkte $x^{(i)}$.

```
In [15]: b, a = lr.coef_
         fig, ax = plt.subplots(figsize=(10,6))
         _ = ax.plot(predicted, residuals, 'o', markeredgecolor='black', color='none',
         alpha=0.4)
         _ = ax.grid(True)
         _ = ax.set_title('Tukey-Ascombe Plot')
         _ = ax.set_ylabel('$r$')
         _ = ax.set_xlabel('$\hat y$')
```





Tukey-Ascombe Plot

Wie es auch ausschauen könnte ..

- ein erkennbares Muster / ein Trend deutet darauf hin, dass die Modellannahmen nicht erfüllt sind.
- bei einem linearen Trend (a) hilft eine log-Transformation der abhängigen Variablen
- bei einer Quadratwurzelabhängigkeit (b) hilft eine Wurzel-Transformation
- allenfalls müsste auch eine gewichtete Regression in Betracht gezogen werden ..

Verifikation unserer Optimierung mit `scikit-learn`

```
In [16]: sklr = skLinearRegression(fit_intercept=False)
         sklr.fit(X, y)
         sklr.coef_
```

```
Out[16]: array([-0.01758912,  1.00570226])
```

Die Methode `score()` berechnet nun (im Fall von Regressionsmethoeden) das Bestimmtheismass `R^2` für ein *Input-Output*-Paar:

```
In [17]: sklr.score(X, y)
```

```
Out[17]: 0.8925479905630459
```

Multiple lineare Regression

Bei der multiplen linearen Regression verfügen wir über mehr als eine *Input*-Variable $x^{(i)} = (x_1^{(i)}, .., x_p^{(i)})$. Das lineare Modell und die Fehlerfunktion für mehrere *Input*-Variablen haben wir bereits eingeführt.

Für diese gilt es nun ein Minimum zu finden.

Es ist dabei wichtig zu bemerken, dass einfache lineare Regression auf einzelnen *Input*-Variablen nur in einem Spezialfall die gleiche Lösung hat wie 'kombinierte' Optimierung. Nämlich dann wenn die *Input*-Variablen orthogonal zueinander sind, das heisst unkorreliert sind.

Für das **iterative Verfahren** berechnet sich der Gradient gleich wie bei der einfachen linearen Regression. Somit lässt sich eine Problemstellung multipler linearer Regression mit dem bereits implementierten Algorithmus lösen.

Wir betrachten nun aber noch eine weitere Möglichkeit:

Line Search

Bisher haben wir eine feste Schrittlänge η verwendet. Es leuchtet ein, dass abhängig von der Umgebung vom Punkt $f(x^{(k)})$ auf der Fläche $f(x)$ mal grössere und mal kleinere Schritte sinnvoll wären.

Tatsächlich gibt es zahlreiche Methoden zur Optimierung der Schrittgrösse $\eta^{(k)}$. Zwei weitere Möglichkeiten sind *Exact Line Search* und *Backtracking Line Search*.

Exact Line Search

Man kann versuchen vom momentanen Punkt und der zuvor bestimmten Richtung entlang des Gradienten einen Schritt so gross zu nehmen, dass f am neuen Punkt kleinstmöglich ist:

$$\eta^{(k)} = argmin_{s \geq 0} f(x - s\nabla f) \tag{14}$$

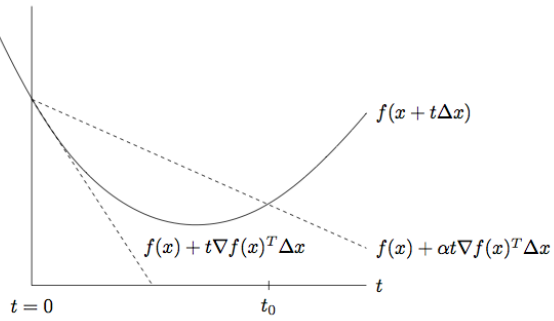
Dieser Ansatz nennt sich aus offensichtlichen Gründen *Exact Line Search*. Unter Umständen lässt sich das Problem in Gleichung 14 analytisch lösen, wodurch dieser Ansatz sehr schnell werden kann. Allgemein ist dies aber nicht der Fall und ein zweites Optimierungsverfahren müsste erst die optimale Schrittlänge errechnen bevor ein Schritt im Hauptverfahren unternommen werden kann. Dies kann sich trotzdem unter Umständen lohnen. Wir behandeln diesen Ansatz aber hier nicht mehr weiter.

Backtracking Line Search

Ein weiterer Ansatz ist *Backtracking Line Search*. Hier suchen wir mit einem kurzen Algorithmus entlang der Richung des negativen Gradienten:

- $\eta = 1$
- solange $f(x - \eta \nabla f(x)) > f(x) - \alpha \eta \nabla f(x)^T \nabla f(x)$: $\eta := \beta \eta$

Zeigen, dass der Algorithmus funktioniert, kann man für $\alpha \in (0, 0.5)$ und $\beta \in (0, 1)$. Typischerweise wählt man $\alpha \in (0.01, 0.3)$ und $\beta \in (0.1, 0.8)$



Backtracking Line Search

Illustration zu *Backtracking Line Search* Für *Gradient Descent* gilt im obigen Bild $\Delta x = -\nabla f(x)$, t entspricht in unserer bisherigen Notation η.
Die obere gestrichelte Linie entspricht dem Abbruchkriterium für die Suche nach der Schrittlänge im obigen Algorithmus.

```
In [18]: class LinearRegressionGradientDescent(object):
             '''
             Implements a Gradient Descent Optimization for a Simple
             Linear Regression Problem.
             '''

             def __init__(self, init_coef=(0., 0.), epsilon=0.00001, maxsteps=1000,
                          stepsize=0.001, linesearch='fixed', alpha=0.1, beta=0.7):
                 '''
                 Gradient Descent Optimizer for Linear Regression.
                 '''
                 self.init_coef = init_coef
                 self.epsilon = epsilon
                 self.maxsteps = maxsteps
                 self.linesearch = linesearch
                 self.stepsize = stepsize
                 self.alpha = alpha
                 self.beta = beta
```

```
        self.coef_ = np.array(self.init_coef)
        self._nsteps = 0
        self.steps = [self.coef_.copy(), ]
        self.cost_function_ = []

        print('LinearRegressionGradientDescent(init_coef={}, epsilon={},\n
maxsteps={},'.format(
                    self.init_coef, self.epsilon, self.maxsteps) +
                ' linesearch={}, stepsize={}, alpha={}, beta={})\n'.format(
                    self.linesearch, self.stepsize, self.alpha, self.beta))


    def fit(self, X, y, verbose=False):
        '''
        Fits the coefficients beta of a linear regression problem
        to the dataset (X, y).
        '''

        normgrad = []
        neg_grad = -self.least_squares_gradient(self.coef_, X, y)
        normgrad.append(np.linalg.norm(neg_grad))

        self.cost_function_.append(self.cost_function(X, y, self.coef_))

        while not self._do_stop(X, y):

            if self.linesearch == 'fixed':
                self.coef_ += self.stepsize*neg_grad

            elif self.linesearch == 'backtracking':
                stepsize = self.backtracking_linesearch(X, y)
                self.coef_ += stepsize*neg_grad

            neg_grad = -self.least_squares_gradient(self.coef_, X, y)
            normgrad.append(np.linalg.norm(neg_grad))

            if verbose:
                print('step {s} : neg grad = {ng}, normgrad: {nog},  coef_ =
{c}, stepsize = {ss}'.format(
                        s=self._nsteps, ng=neg_grad, c=self.coef_,
                        nog=np.linalg.norm(neg_grad),
                        ss=self.stepsize if self.linesearch == 'fixed' else
stepsize ))

            self._nsteps += 1
            self.cost_function_.append(self.cost_function(X, y, self.coef_))
            self.steps.append(self.coef_.copy())

        self.normgrad_ = np.array(normgrad)
```

```
            self.cost_function_ = np.array(self.cost_function_)

            print('Done fitting!')

            print('n steps : ', self._nsteps)
            print('norm gradient : ', self.normgrad_[-1])

            return self


        def _do_stop(self, X, y):
            grad = self.least_squares_gradient(self.coef_, X, y)
            tiny = np.linalg.norm(grad) < self.epsilon
            manysteps = self._nsteps > self.maxsteps

            return (tiny or manysteps)


        def predict(self, X):
            return X.dot(self.coef_)


        @staticmethod
        def least_squares_gradient(coef, X, y):
            '''
            Calculates the least squares gradient at position coef
            for the dataset (X, y).
            '''
            grad = []
            for idx in range(X.shape[1]):
                grad.append(((y - coef.dot(X.T))*X[:, idx]).sum())
            return -2*np.array(grad)


        ##########################################################################
    ##############


        def score(self, X, y):
            '''
            '''
            residual = y - self.predict(X)
            return 1. - ((residual -
    residual.mean())**2).sum()/((y-y.mean())**2).sum()


        def backtracking_linesearch(self, X, y):
            eta = 1
            alpha = self.alpha
```

```
            beta = self.beta
            grad = self.least_squares_gradient(self.coef_, X, y)
            fxs = self.cost_function(X, y, self.coef_-eta*grad)
            fxpa = self.cost_function(X, y, self.coef_) - alpha*grad.T.dot(grad)

            while fxs > fxpa:
                eta *= beta
                fxs = self.cost_function(X, y, self.coef_-eta*grad)
                fxpa = self.cost_function(X, y, self.coef_) -
        alpha*eta*grad.T.dot(grad)
            return eta


        def cost_function(self, X, y, beta):
            residual = y - X.dot(beta)
            return (residual**2).sum()
```

```
In [19]: lr_bt = LinearRegressionGradientDescent(init_coef=(-1., -1.), epsilon=0.00001,
                                                 maxsteps=10000,
         linesearch='backtracking')
         lr_bt = lr_bt.fit(X, y)

LinearRegressionGradientDescent(init_coef=(-1.0, -1.0), epsilon=1e-05,
 maxsteps=10000, linesearch=backtracking, stepsize=0.001, alpha=0.1, beta=0.7)

Done fitting!
n steps :  10001
norm gradient :  1.49833392804e-05
```

Normalengleichung

Wir betrachten hier nun noch die Möglichkeit, das Problems ein Minimum für $J(\beta)$ zu finden, analytisch zu lösen:

Wir können die Fehlerfunktion $J(\beta)$ etwas umschreiben:

Residuum-Vektor Transponieren für Matrixmultiplikation, was der Quadrierten Summe entspricht!!

$$J(\beta) = \sum_{i=1}^{N} (y^{(i)} - l(\beta, x^{(i)}))^2 = (y - X\beta)^T (y - X\beta) \tag{15}$$

Wenn wir diese Funktion nach den Koeffizienten β ableiten finden wir folgendes:

$$\frac{\partial J(\beta)}{\partial \beta} = -2X^T(y - X\beta) \tag{16}$$

Zum Finden des Minimums können wir diese erste Ableitung gleich 0 setzen:

$$-2X^T(y - X\beta) = 0 \tag{17}$$

Und finden dann die exakte Lösung:

$$(X^T X)^{-1} X^T y = \hat{\beta} \tag{18}$$

Beachten sie, dass

- wir $\hat{\beta}$ schreiben, um zu unterstreichen, dass es sich um einen 'Schätzwert' der wahren Koeffizenten handelt.
- die quadratische Matrix $X^T X$ nur invertierbar ist wenn sie regulär ist, das heisst ihr Rang gleich ihrer Anzahl Zeilen bzw. Spalten ist.
- die Berechnung der Inversen von $X^T X$ für grosse Datensätze sehr aufwendig sein kann und sich dieser Ansatz deshalb für grosse Datensätze nicht eignet.

- die Inverse einer Matrix A, sodass $A^{-1} A = \mathbf{1}$, mit `np.linalg.inv(A)` berechnet werden kann.