Optimization for Machine Learning

Nick Henderson, AJ Friend (Stanford University)
Kevin Carlberg (Sandia National Laboratories)

August 14, 2018

Model fitting

Outline

Model fitting

Linear least squares

Linear least squares, non-linear data

Linear least squares in matrix–vector form

Examples

Nonlinear least squares

Beyond least squares

Deep Feedforward Networks

Stochastic methods

Notation

The notation between these worlds is not consistent

- Optimization
    - f: optimization objective function
    - x: optimization variables
- Machine-learning regression (this set of slides)
    - f: regression function mapping inputs to outputs
    - x: regression-model inputs (i.e., independent variable)
    - y: regression-model outputs (i.e., response variable)
    - ϕ: optimization objective function (i.e., loss function)
    - β or θ: optimization variables (i.e., model parameters)

Least-squares regression

- A particular application of model fitting with many applications
- **Goal**: find a model that best fits training data in the least-squares sense
- Illuminates the connection between **unconstrained optimization** and **statistics/machine learning**
- We will use the following iPython notebooks
  - `least-squares.ipynb`
  - `polynomial-fit.ipynb`
  - `smooth.ipynb`
  - `huber.ipynb`

Linear least squares

Outline

Problem set up

- **Given**: some data (i.e., training set)

x: independent variable	y: response variable
0.0	0.46
0.11	0.31
0.22	0.38
0.33	0.39
0.44	0.65
0.56	0.40
0.67	0.87
0.78	0.69
0.89	0.87
1.0	0.88

- **Goal**: construct a model that can predict y from x

Where might these data come from?

x: independent variable	y: response variable
height	weight
square feet	price of home
device property	failure rate
stock market return	individual asset return

- Regression can be applied regardless of the origin of the data!

Regression: Approach

***Goal**: construct a model that can predict y from x*

- In general, we *do not* know the mathematical model characterizing the underlying process that actually generated the data
- So, we *assume* that the data were generated from a model comprising the sum of a (deterministic) function and (stochastic) iid Gaussian noise:

  $$y_i = f_{\text{true}}(x_i) + \sigma \cdot \epsilon_i$$

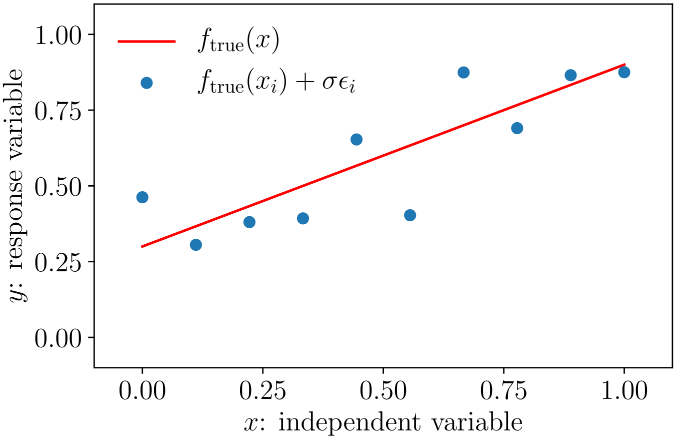
  with $f_{\text{true}}(x_i)$ unknown and $\epsilon_i \sim N(0, 1)$
- We then aim to construct $f(x)$ such that $f(x) \approx f_{\text{true}}(x)$ in some sense
- This is known as *regression* and is performed via optimization
  - *objective function*: sum of squared errors $\sum_i (y_i - f(x_i))$ over a training set
  - *optimization variables*: parameters within the assumed form of $f(x)$
- Then, we can make predictions $y(x) \approx f(x)$ for new values of x.

Follow along in Python

- See `least-squares.ipynb`
- In this case, we have set $f_{\text{true}}(x) = m_{\text{true}} \cdot x_i + b_{\text{true}}$ and $\sigma = 0.1$
  - $m_{\text{true}} = 0.6$
  - $b_{\text{true}} = 0.3$
- Run the first three cells of `least-squares.ipynb`
- Python code to generate data (in second cell):

```
np.random.seed(1)
m = 0.6
b = 0.3
sigma = .1
x = np.linspace(0,1,10)
y = m*x + b + sigma*np.random.standard_normal(x.shape)
```

Let's plot the data



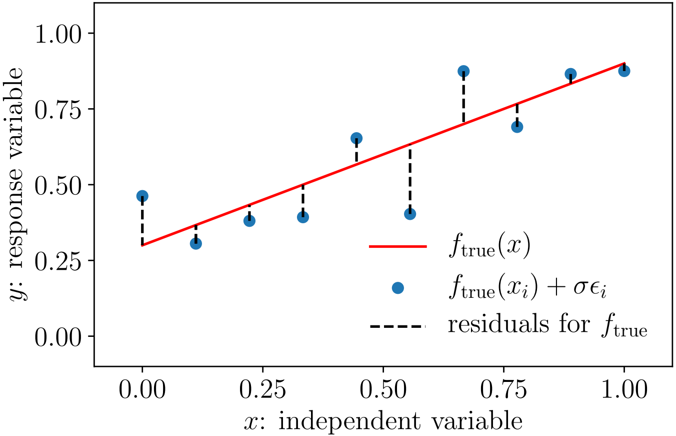
The residuals

- Any given data point will result in some error or **residual**

$$r_i = f(x_i) - y_i$$

- Due to the Gaussian noise, $y_i \neq f_{\text{true}}(x_i)$. Thus, the true function $f_{\text{true}}(x)$ will yield residuals

The residuals for $f_{\text{true}}(x)$



Linear regression in one dimension

- In linear regression, we enforce the regression function $f(x)$ to be a linear function

$$f(x) = m \cdot x + b$$

  - x is the input variable (i.e., feature, independent variable)
  - y is the output variable (i.e., response, dependent variable)
  - regression function has two parameters: the slope m and the y-intercept b
- We can encode this parameter dependence as

$$f(x; m, b) = m \cdot x + b$$

  - The semi-colon separates the independent variable x from the parameters (m, b)
- **Note**: the form of $f(x)$ need not match the (generally unknown) form of $f_{\text{true}}(x)$. We are lucky if this happens!

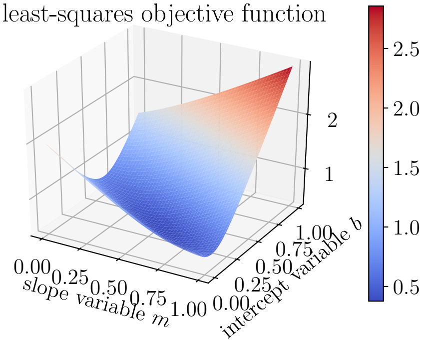
Fit the model via optimization

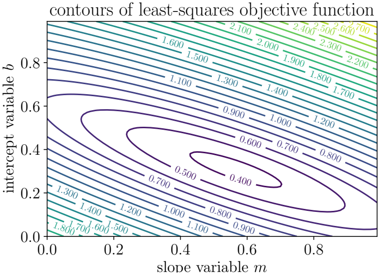
- To fit this linear model to data, we construct an optimization problem that aims to minimize the residuals

$$\text{minimize} \quad \phi(m,b) = \tfrac{1}{2}\textstyle\sum_{i=1}^{n} r_i^2 = \tfrac{1}{2}\sum_{i=1}^{n}(f(x_i;m,b) - y_i)^2$$

  - *Optimization objective function*: residual sum of squares (RSS)
    - Note that this objective function comprises a sum of contributions from each training example
  - *Optimization variables*: parameters m and b
- If the true underlying model actually is $y_i = m_{\text{true}} \cdot x_i + b_{\text{true}} + \sigma \cdot \epsilon_i$ with ϵ_i mean-zero Gaussian, then m and b are the maximum-likelihood estimates of m_{true} and b_{true}

Objective function





- The objective function $\phi(m, b)$ is appears to be convex (it is!)
- The global minimum occurs around $m^\star \approx 0.6$ and $b^\star \approx 0.35$

Optimizing by hand

Recall the sufficient conditions for (unconstrained) optimality:

1. $\nabla \phi(m^\star, b^\star) = 0$
2. $\nabla^2 \phi(m^\star, b^\star) \succ 0$. **This holds everywhere!**
    - The objective function is strongly convex
    - This simplifies our lives greatly, as we only need to find a stationary point satisfying condition 1
    - This is one reason why convex optimization is so nice!

Let's compute $m^\star$ and $b^\star$ such that that the first condition holds.

Compute gradient analytically and set to zero

Analytical gradient computation:

$$\frac{\partial \phi}{\partial m} = \frac{1}{2}\sum_{i=1}^{n} \frac{\partial}{\partial m}(m \cdot x_i + b - y_i)^2 = m\sum x_i^2 + b\sum x_i - \sum x_i y_i$$

$$\frac{\partial \phi}{\partial b} = \frac{1}{2}\sum_{i=1}^{n} \frac{\partial}{\partial b}(m \cdot x_i + b - y_i)^2 = m\sum x_i + nb - \sum y_i$$

Set analytical gradient to zero and obtain a system of equations:

$$\frac{\partial \phi}{\partial m} = 0$$

$$\frac{\partial \phi}{\partial b} = 0$$

Solution

$$m = \frac{\sum x_i y_i - \frac{1}{n} \sum x_i \sum y_i}{\sum x_i^2 - \frac{1}{n} (\sum x_i)^2}$$

$$b = \frac{\sum y_i - m \sum x_i}{n}$$

Let's look at m

Something looks nice here:

$$m = \frac{\sum x_i y_i - \frac{1}{n} \sum x_i \sum y_i}{\sum x_i^2 - \frac{1}{n} (\sum x_i)^2}$$

Multiply both numerator and denominator by $1/n$:

$$m = \frac{\frac{1}{n} \sum x_i y_i - \frac{1}{n} \sum x_i \frac{1}{n} \sum y_i}{\frac{1}{n} \sum x_i^2 - (\frac{1}{n} \sum x_i)^2}$$

We see sample covariance and variance here!

$$m = \frac{\mathsf{cov}(X, Y)}{\mathsf{var}(X)}$$

Let's solve in Python!

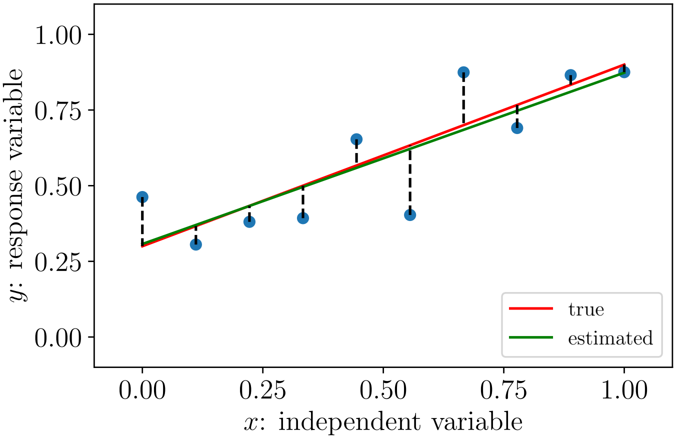
Code:

```
# solve via numpy covariance function
A = np.vstack((x,y))
V = np.cov(A)
m_est = V[0,1] / V[0,0]
b_est = (y.sum() - m_est*x.sum()) / len(x)
print(m_est)
print(b_est)
```

Result:

```
m_est = 0.56604 (true value = 0.6)
b_est = 0.30727 (true value = 0.3)
```

Look at the plot



Solve in CVXPY

Remember the optimization problem: minimize $\frac{1}{2}\sum_{i=1}^{n}(m \cdot x_i + b - y_i)^2$

We can write this directly in CVXPY:

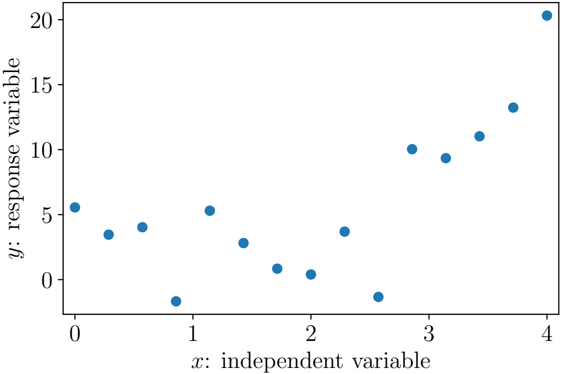
```
from cvxpy import *
# Construct the problem.
m_cvx = Variable()
b_cvx = Variable()
objective = Minimize(sum_squares(m_cvx*x + b_cvx - y))
prob = Problem(objective)
# The optimal objective is returned by prob.solve().
result = prob.solve()
```

`m_cvx.value = 0.56604, b_cvx.value = 0.30727`

Linear least squares, non-linear data

Outline

What about these data?

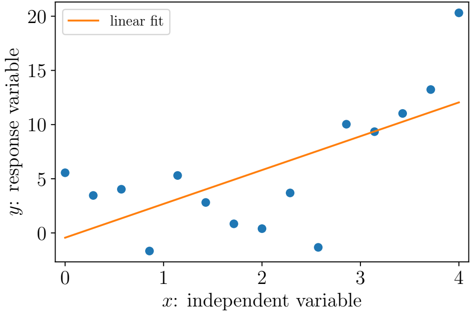


- Here, $f_{\mathrm{true}}(x) = m_{\mathrm{true}} \exp(x) + b_{\mathrm{true}}$, which we do not know *a priori*
- We just have access to the data!

We can fit a linear model

- Given our ignorance of f_{true}, we can fit a linear model

$$f(x; m, b) = m \cdot x + b,$$



- This yields an objective-function value of $\phi(m, b) = 283.63$

We can also fit an exponential model

- If we think that the underlying model may be exponential, we can also try

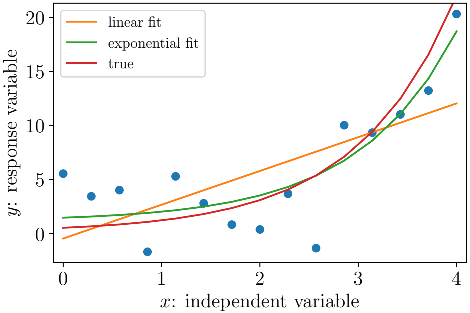
$$f(x; m, b) = m \cdot \exp(x) + b$$

- **Note**: model is still *linear in the parameters* m *and* b!
- Thus, we can again apply least squares variable.

CVXPY code:

```
m = Variable()
b = Variable()
objective = Minimize(sum_squares(m*np.exp(x) + b - y))
prob = Problem(objective)
result = prob.solve()
```

Result



- This yields a *smaller* objective-function value of $\phi(m, b) = 122.80$, implying that the exponential model better fits the training data
- We have to be careful about this as a measure of performance, as we can overfit training data (assessing generalization error on a test set required)

Linear least squares in matrix–vector form

Outline

- Model fitting
- Linear least squares
- Linear least squares, non-linear data
- Linear least squares in matrix–vector form
- Examples
- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

More variables

- We've been talking about least squares with *one* independent variable ($n = 1$)
- Utility of linear models is increased when we incorporate *multiple* variables ($n > 1$)
- For this discussion, we need vectors and matrices!

Vectors

- A *vector* is an array of numbers
- Vectors have a length or number of elements, usually denoted n
- By default, we think of vectors as numbers being stacked in a column

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Vector operations

- Multiplication by scalars

$$\alpha x = \begin{bmatrix} \alpha x_1 \\ \alpha x_2 \\ \vdots \\ \alpha x_n \end{bmatrix}$$

- Vector addition

$$x + y = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

Column and row vectors

By convention, we consider vectors to be oriented as columns:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Sometimes we want a row vector. We use the transpose operation for this:

$$x^T = \begin{bmatrix} x_1 \; x_2 \; x_3 \; \cdots \; x_n \end{bmatrix}$$

Vector inner product

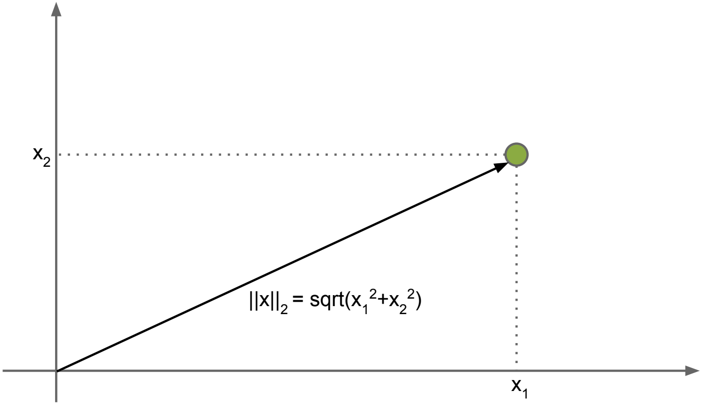
- x and y are both vectors with length n
- Also known as a dot product

$$x^T y = x \cdot y = \sum_{i=1}^{n} x_i y_i$$

- Inner product between x and x

$$x^T x = x \cdot x = \sum_{i=1}^{n} x_i^2$$

Vector norm



Vector norm

- The magnitude of a vector can be measured using a p-norm, defined as

$$||x||_p = (\sum_{i=1}^{n} (x_i)^p)^{1/p}$$

- The 2-norm $(p = 2)$ is the most widely used p-norm. This case corresponds to the sum of squares of the vector elements:

$$||x||_2 = \sqrt{\sum_{i=1}^{n} (x_i)^2} = \sqrt{x^T x}$$

- The 1-norm $(p = 1)$ is also a common p-norm. This case corresponds to the sum of the absolute values of the vector elements:

$$||x||_1 = \sum_{i=1}^{n} |x_i|$$

Least-squares objective

- Let's say we have a data set (a_i, y_i)
- a is now the independent variable and y is the response variable
- The residuals arising from the linear model are $r_i = y_i - m \cdot a_i - b$
- This can be written in vector form as $r = y - m \cdot a - b \cdot \mathbf{1}$
  - r, y, and a are column vectors
  - $\mathbf{1}$ is a column vector of ones
- The optimization formulation to find parameters m and b is

$$\text{minimize} \quad \frac{1}{2}\sum_{i=1}^{n}(m \cdot a_i + b \cdot 1 - y_i)^2 = \frac{1}{2}||m \cdot a + b \cdot 1 - y||_2^2 = \frac{1}{2}||r||_2^2$$

Matrices

- A *matrix* is an array of numbers with height m and width n
- We write matrix elements with lower case letters and indexed by row then column
  - a_{ij} refers to element in row i and column j

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{33} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Matrix-vector products

- Say A is an $m \times n$ matrix and x is an n-vector
- Length of vector must match the second dimension of matrix
- The product is: $y = Ax$
- y is an m-vector, the number of rows in A
- Formula for the product: $y_i = \sum_{j=1}^{n} a_{ij} x_j$
- Example:

$$y = Ax = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 \end{bmatrix}$$

Matrix-matrix product

- A is an $m \times n$ matrix
- B is an $n \times p$ matrix
- $C = AB$ is the product of A and B and is an $m \times p$ matrix
- The elements of C may be computed with

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- Size of the inner dimensions of the matrices must match

Matrix-vector product revisited

$$y = Ax = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 \end{bmatrix}$$

- we can consider the column vector to be a $n \times 1$ matrix
- matrix-vector product is special case of matrix-matrix product
- the inner product between vectors is a product between matrices of size $1 \times n$ and $n \times 1$

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

Matrix-vector form for least squares

$$\text{minimize} \quad \tfrac{1}{2}\textstyle\sum_{i=1}^{n}(m \cdot a_i + b - y_i)^2$$

- Pack data from the independent variable and a constant into matrix A and model parameters into vector x:

$$A = \begin{pmatrix} a & \mathbf{1} \end{pmatrix} = \begin{pmatrix} a_1 & 1 \\ a_2 & 1 \\ \vdots & \vdots \\ a_n & 1 \end{pmatrix}, \qquad x = \begin{bmatrix} m \\ b \end{bmatrix}$$

- The residuals arising from the linear model are now:

$$r = y - m \cdot a - b \cdot \mathbf{1} = y - Ax$$

Standard form for least squares

$$\text{minimize} \quad \tfrac{1}{2}||Ax - b||_2^2$$

In the context of model fitting:

- A is the matrix that contains data from independent variables
- b is the vector containing response data (y previously)
- x is the vector of model parameters
- The optimization problem above is solved to find x^*, the parameters that minimize the sum of squared residuals
- For each item of data, we have the equation where a_i^T is row i of A

$$b_i - a_i^T x = r_i$$

Notation from statistics

$$\text{minimize} \quad \frac{1}{2}||\mathbf{y} - \mathbf{X}\beta||_2^2$$

The statistics community often uses different notation:

- $\mathbf{X}$ is the matrix that contains data from independent variables
- $\mathbf{y}$ is the vector holding response data
- β is the vector of model parameters

CVXPY for least squares

```
# generate input and response data
np.random.seed(1); n = 10 # number of data points
input_data = np.linspace(0,1,n)
response_data = 0.6*input_data + 0.3 + 0.1*np.random.standard_normal(n)
# least-squares matrix and vector
A = np.vstack([input_data,np.ones(n)]).T; b = response_data
# CVX problem
x = Variable(A.shape[1])
objective = Minimize(sum_squares(A*x - b))
prob = Problem(objective); result = prob.solve()
# get value & print
x_star = np.array(x.value)
print('slope = {:.4}, intercept = {:.4}'.format(x_star[0,0],x_star[1,0]))
```

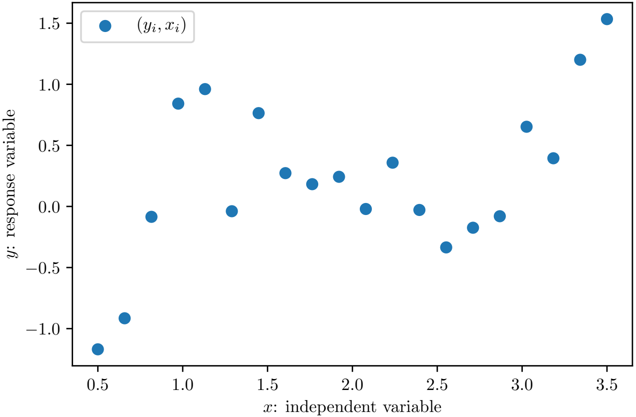
```
slope = 0.566, intercept = 0.3073
```

Examples

Outline

- Model fitting
- Linear least squares
- Linear least squares, non-linear data
- Linear least squares in matrix–vector form
- Examples
- Nonlinear least squares
- Beyond least squares
- Deep Feedforward Networks
- Stochastic methods

What about these data?



We can try a polynomial model

- Have m data points (u_i, y_i)
- The model

$$y \approx p(x) = \beta_1 + \beta_2 x + \beta_3 x^2 + \cdots + \beta_n x^{n-1}$$

- x is the independent variable
- y is the response variable
- β_i are the model parameters and coefficients of the polynomial
- Model is linear in the parameters! We can use least squares!

Polynomial model

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \end{bmatrix} \approx \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ 1 & x_4 & x_4^2 & \dots & x_4^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_n \end{bmatrix}$$

$$y \approx A\beta$$

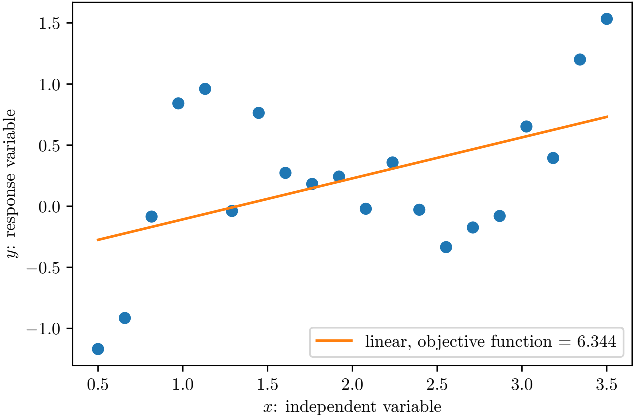
- A is known as the *Vandermonde matrix*

Solve with CVXPY

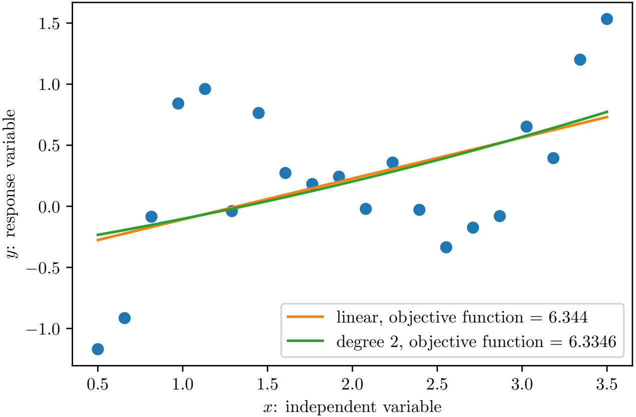
- See `polynomial-fit.ipynb`

```
def cvxpy_poly_fit(x,y,degree):
    # construct data matrix
    A = np.vander(x,degree+1)
    b = y
    beta_cvx = Variable(degree+1)
    # set up optimization problem
    objective = Minimize(sum_squares(A*beta_cvx - b))
    constraints = []
    # solve the problem
    prob = Problem(objective,constraints)
    prob.solve()
    # return the polynomial coefficients
    return np.array(beta_cvx.value)
```

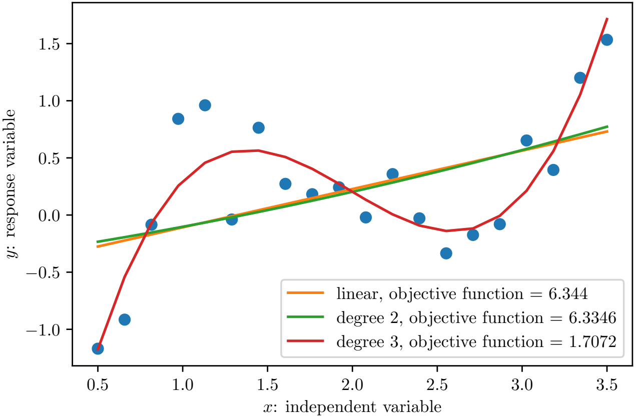
Linear fit


1.5
1.0
0.5
0.0
−0.5
−1.0
y: response variable
0.5
1.0
1.5
2.0
2.5
3.0
3.5
x: independent variable
linear, objective function = 6.344

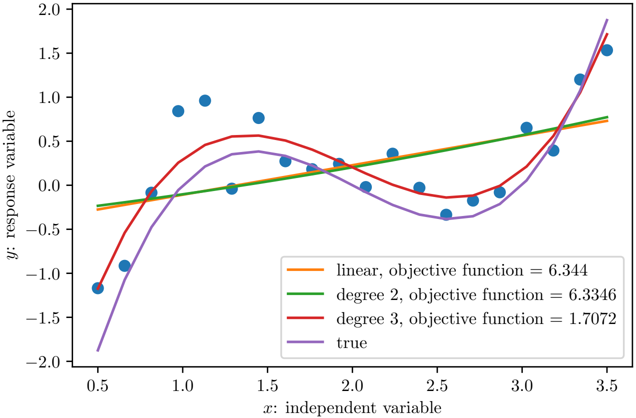
Quadratic fit



Cubic fit


1.5
1.0
0.5
0.0
−0.5
−1.0
y: response variable
0.5
1.0
1.5
2.0
2.5
3.0
3.5
x: independent variable
linear, objective function = 6.344
degree 2, objective function = 6.3346
degree 3, objective function = 1.7072

True model

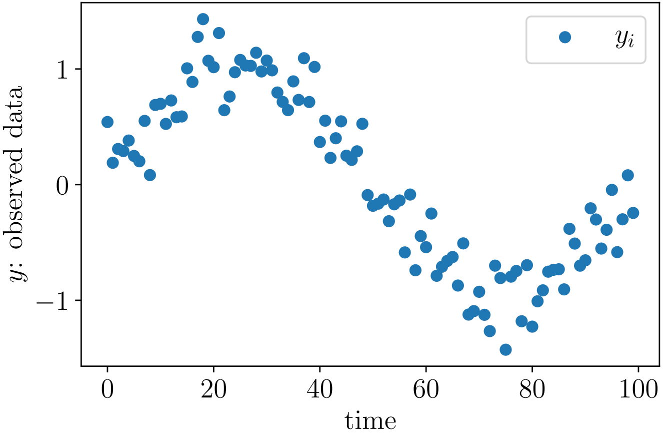


- This was the true (generally unknown) model that generated the data

Example: time series smoothing

- See `smooth.ipynb`
- m noisy observations at regular interval, $y \in \mathbf{R}^m$ (discretized curve)
- Modeling approach
  - Don't have a model for the curve (linear, polynomial, . . .)
  - Do believe that the curve should be "smooth"
- **Idea**: find $x \in \mathbf{R}^m$ which is close to y, but also penalized for being nonsmooth
  - Here, the number of optimization variables n is equal to number of data points m
  - We are not using any underlying functional form

Time series data



Optimization problem

- Want each x_i to be close to each y_i
- Want vector x to represent a smooth function
- Optimization problem

$$\text{minimize } ||x - y||_2^2 + \rho \cdot \text{penalty}(x)$$

- Use penalty function to encourage smoothness
- Use parameter ρ to trade off
  1. fit to data (ρ small), and
  2. smoothness (ρ large)

Measurement of smoothness

- define smoothness to be a curve whose slope does not change much
- change in slope (of a smooth curve) given by the second derivative
- change in slope of a discretized curve can be approximated as Dx via second-order finite differences, where

$$D = \begin{pmatrix} 1 & -2 & 1 & 0 & \dots & & & 0 \\ 0 & 1 & -2 & 1 & 0 & \dots & & 0 \\ 0 & 0 & 1 & -2 & -1 & 0 & \dots & 0 \\ \vdots & & & & & & & \end{pmatrix}$$

Least squares model

- x close to the data if $\|x-y\|_2^2$ is small
- x is smooth if $\|Dx\|_2^2$ is small
- solve the least-squares problem

$$\text{minimize} \quad \|x-y\|_2^2 + \rho\|Dx\|_2^2$$

- ρ trades-off between fitness to the data and smoothness
  - small ρ: more model flexibility (more effective degrees of freedom)
  - large ρ: less model flexibility (fewer effective degrees of freedom)

Standard form

Model:

$$\text{minimize} \quad \|x - y\|_2^2 + \rho\|Dx\|_2^2$$

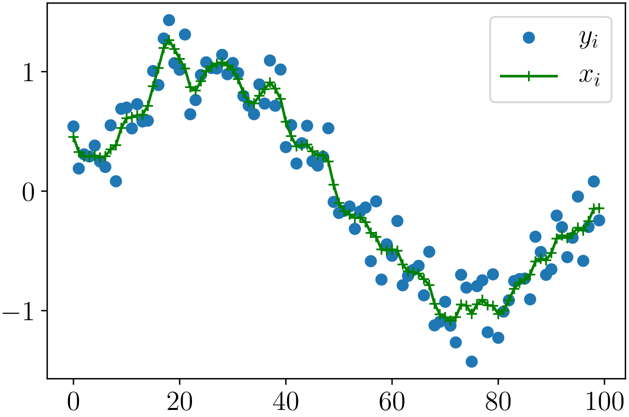
Standard form:

$$\text{minimize} \left\| \begin{pmatrix} I \\ \rho D \end{pmatrix} x - \begin{pmatrix} y \\ 0 \end{pmatrix} \right\|_2^2$$

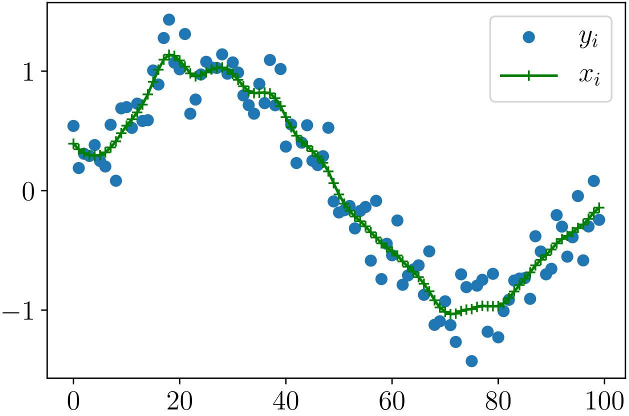
Solve the problem in CVXPY

```
# get second-order difference matrix
D = diff(n, 2) # user-defined function
rho = 1
# construct and solve problem
x = cvx.Variable(n)
cvx.Problem(cvx.Minimize(cvx.sum_squares(x-y)
                         +rho*cvx.sum_squares(D*x))).solve()
x = np.array(x.value).flatten()
```

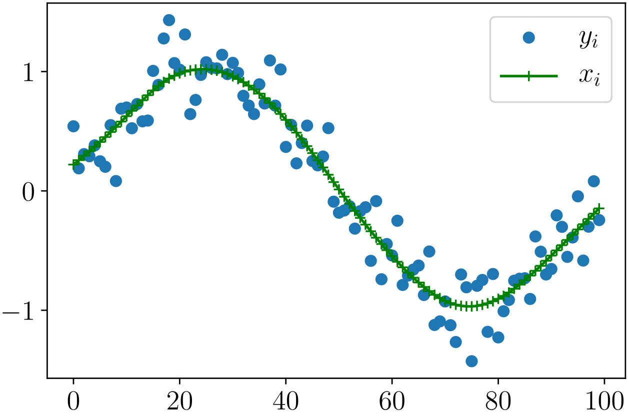
$\rho = 1$



$\rho = 10$



$\rho = 1000$



Nonlinear least squares

Outline

Model fitting

Linear least squares

Linear least squares, non-linear data

Linear least squares in matrix–vector form

Examples

Nonlinear least squares

Beyond least squares

Deep Feedforward Networks

Stochastic methods

Nonlinear least squares

Linear least squares:

1. Employ a model that is *linear in the parameters*

$$f(x;\beta) = \sum_{i=1}^{n} \beta_i g_i(x)$$

   - g_i can be nonlinear in x (e.g., polynomials, expontial)
   - if g_i is also linear in x, then this is 'linear regression'

2. Minimize the residual sum of squares (RSS)

$$\text{minimize} \quad \phi(\beta) = \tfrac{1}{2}\sum_{i=1}^{m} r_i^2 = \tfrac{1}{2}\sum_{i=1}^{m}(f(x_i;\beta) - y_i)^2$$

Nonlinear least squares:

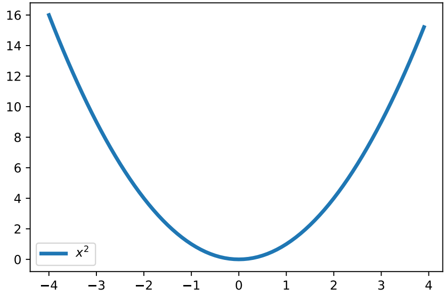
1. Employ a model $f(x;\beta)$ that is *nonlinear in the parameters* β
2. Minimize the same objective: residual sum of squares (RSS)
   - Again equivalent to maximum likelihood if additive Gaussian noise
   - Algorithms: line-search (Gauss–Newton) and trust-region (Levenberg–Marquardt)

Beyond least squares

Outline

Quadratic loss function

- See `huber.ipynb`
- Least squares employs a quadratic loss function

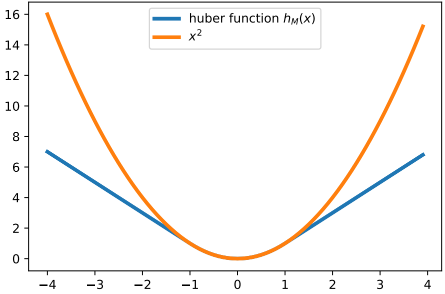


- This function imposes a severe penalty on large values
- As a result, the fit model is very sensitive to outliers (can overfit)
- Can we use a different loss function?

Huber loss function

- The Huber function allows us to better handle outliers in data
  - Usual quadratic loss in interval $[-M, M]$
  - Linear loss for $|x| > M$

$$h_M(x) = \begin{cases} x^2 & |x| \leq M \\ 2M|x| - M^2 & |x| > M \end{cases}$$



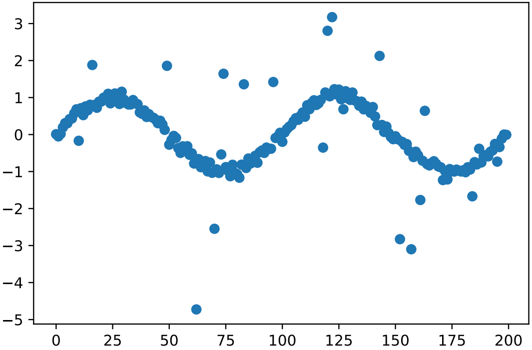
Huber loss function

- This function imposes a less severe penalty on large values
- Let's repeat the curve-fitting example, but include extreme outliers in the data
- Penalize closeness to data with Huber function h_M to reduce influence of outliers:

$$\text{minimize} \quad \textstyle\sum_{i=1}^{n} h_M(x_i - y_i) + \rho \|Dx\|_2^2$$

  - M parameter controls width of quadratic region, or "non-outlier" errors
- This is no longer least squares!
- CVXPY has implemented the Huber loss function

Huber data

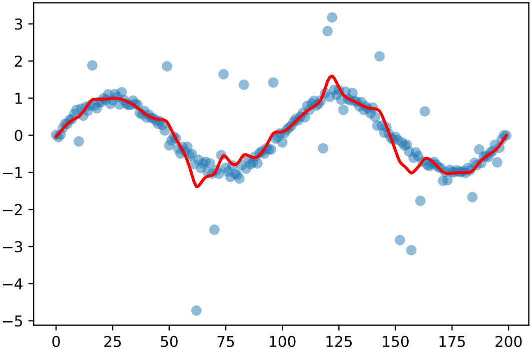


Least-squares smoothing

```
# get second-order difference matrix
D = diff(n, 2)
rho = 20

x = Variable(n)
obj = sum_squares(x-y) + rho*sum_squares(D*x)
Problem(Minimize(obj)).solve()
x = np.array(x.value).flatten()
```

Least-squares smoothing result



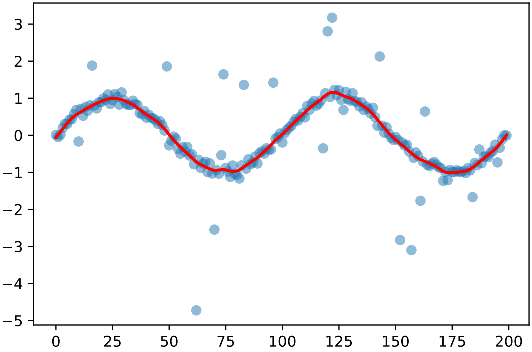
- Model overfits the outliers

Huber smoothing

```
# get second-order difference matrix
D = diff(n, 2)
rho = 20
M = .15 # huber radius

x = Variable(n)
obj = sum_entries(huber(x-y, M)) + rho*sum_squares(D*x)
Problem(Minimize(obj)).solve()
x = np.array(x.value).flatten()
```

Huber smoothing result



- The model is less sensitive to outliers!

Deep Feedforward Networks

Outline

Deep Feedforward Networks

A deep feedforward network defineds a particular model $f(x;\beta)$

- $f(x;\beta) = f^{(3)}(f^{(2)}(f^{(1)}(x;\beta_1);\beta_2);\beta_3)$ is a 'network' (function composition)
    - $f^{(i)}(x;\beta_i)$: function charactering the ith layer with parameters β_i
    - parameters $\beta = (\beta_1, \beta_2, \beta_3) \in \mathbf{R}^n$
- Evaluating f is 'forward propagation': start at the beginning ($f^{(1)}$) and evaluate forward sequentially
- It is 'deep' if there are many composed functions, and thus β is high-dimensional
- f is genearlly nonlinear in the parameters β
- if additive Gaussian noise, then MLE leads to nonlinear least squares
- other loss functions possible (e.g., non-Gaussian noise); then no longer least squares

Deep Feedforward Networks

- Computing the gradient can be done by applying the chain rule, e.g.,

$$\frac{\partial \phi}{\partial \beta_2} = \frac{\partial \phi}{\partial f^{(3)}} \frac{\partial f^{(3)}}{\partial x} \frac{\partial f^{(2)}}{\partial \beta_2}, \quad \frac{\partial \phi}{\partial \beta_1} = \frac{\partial \phi}{\partial f^{(3)}} \frac{\partial f^{(3)}}{\partial x} \frac{\partial f^{(2)}}{\partial x} \frac{\partial f^{(1)}}{\partial \beta_1}$$

- Computing the gradient is referred to as **back propagation**: the chain rule 'propagates' information from the end of the network ($f^{(3)}$) upstream (e.g., to $f^{(1)}$)

Deep Feedforward Networks: optimization challenges in optimization

$$\text{minimize} \quad \phi(\beta) = \tfrac{1}{2}\sum_{i=1}^{m}(f(x_i;\beta) - y_i)^2$$

High-dimensional

- many β parameters n (due to many layers)
- many data points m and (need lots of data to tune many parameters)
- **solution**: stochastic/minibatch methods (e.g., stochastic gradient descent)

Non-convex

- can get trapped in local minima
- **solution**: local minima seem to yield a "low-enough" cost-function value

Ill conditioning

- **solution**: second-order methods (but hard for NNs)

Stochastic methods

Outline

What does 'Big Data' mean for model fitting?

- In model fitting, the objective function is usually composed of a sum of m contributions:

$$\phi(\beta) = \frac{1}{m} \sum_{i=1}^{m} \phi_i(\beta)$$

  - ϕ_i: is the loss associated with the ith training example
  - ϕ: a sampling-based approximation of the expected loss
- 'Big Data' can refer to:
  - many training examples: m large
  - many parameters: n large
  - deep learning falls in this category!
- Specialized methods have been developed for these cases!
  - **stochastic/minibatch methods** (next)
  - distributed optimization (see 'Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers' by Boyd et al.)

Stochastic methods

Here, the gradient is also a sum of m contributions:

$$\nabla\phi(\beta) = \frac{1}{m}\sum_{i=1}^{m}\nabla\phi_i(\beta)$$

- **Batch methods** use this within gradient-based optimization
- **Benefit**: Preserves traditional convergence rates
- **Drawbacks**:
  - Requires accessing all m data points each iteration *(costly)*
  - Many data points are likely *redundant*
- Can we make this less expensive yet still maintain convergence?

Observations:

1. The objective is (usually) just the **sample mean** of the loss function
2. Expectations via Monte Carlo sampling converge slowly (rate $m^{-1/2}$)
3. Exact gradients aren't needed for convergence

Idea: inexpensively approximate the gradient with a *sample* of the data

Stochastic methods

Stochastic methods: compute approximate the gradient as

$$\nabla \phi(\beta) \approx \nabla \phi_i(\beta)$$

- i is a randomly chosen training example
- **Stochastic gradient descent (SGD)**: stochastic approximation to gradient descent:

  $$x_{i+1} = x_k - \alpha_k \nabla \phi_i(\beta)$$

- **Benefits**:
  - each iteration is *much cheaper*
  - often observe faster rate of convergence *as a function of accessed data points*
  - a descent direction *in expectation*, i.e., $\mathbb{E}[\nabla \phi_i(\beta)] = \nabla \phi(\beta)$
- **Drawbacks**
  - slower rate of convergence *as a function of iteration* (sublinear for SGD)
  - observed slowdown as iterations progress due to noisy gradients

SGD performance in practice

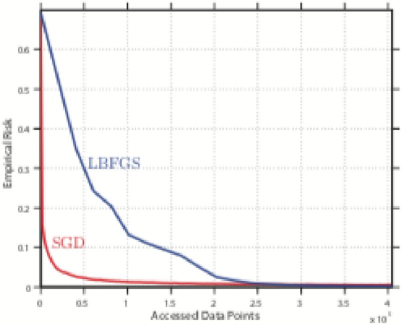


Fig. 3.1 *Empirical risk R_n as a function of the number of accessed data points (ADPs) for a batch L-BFGS method and the SG method (3.7) on a binary classification problem with a logistic loss objective and the RCV1 dataset. SG was run with a fixed stepsize of $\alpha = 4$.*

Reference: Bottou, L., Curtis, F.E. and Nocedal, J., 2018. Optimization methods for large-scale machine learning. SIAM Review, 60(2), pp.223-311.

Improving the convergence rate of stochastic methods

Noise reduction: reduce variance gradient estimate

- *Dynamic sampling*: use **minibatch** estimates of the gradient at iteration k

  $$\nabla\phi(\beta) \approx \frac{1}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} \nabla\phi_i(\beta),$$

  where the minibatch size $|\mathcal{S}_k|$ increases with k.
- *Gradient aggregation*: reuse recently computed gradient information
  - *Example*: stochastic variance reduced gradient (SVRD):

    $$\nabla\phi(\beta) \approx \nabla\phi_i(\beta) - (\nabla\phi_i(\bar{\beta}) - \nabla\phi(\bar{\beta}))$$

  - $\bar{\beta}$: variables the last time the true batch gradient was computed

Improving the convergence rate of stochastic methods

Second-order methods: use sampled Hessian information

- **Subsampled Hessian-Free Newton Methods**: minibatch estimate of the Hessian

$$\nabla^2 \phi(\beta) \approx \frac{1}{|\mathcal{S}_k^H|} \sum_{i \in \mathcal{S}_k^H} \nabla^2 \phi_i(\beta)$$

  - Can also enforce positive definiteness via subsampled Gauss–Newton approximations
- **Subsampled Quasi-Newton Methods**:
  - typical quasi-Newton methods with stochastic estimates of the gradient