aiLib Documentation

Release 1.0

Matthias Baumgartner

March 06, 2012

CONTENTS

Welcome to the aiLib documentation.

The **aiLib** is a collection of algorithms used in the broad field of artificial intelligence. This mainly includes machine learning topics and a lot of statistics.

The first two chapters of this documentation are introductory. The first chapter (*Introduction*) covers information about the documentation and a statistics formulary. The second chapter (*core — Core functionality*) explains basic library routines and documents their interfaces. The other chapters describe the implemented algorithms. Most documents are structured alike. After some introduction, the theoretical background is explained. This part often covers a mathematical derivation or abstract algorithm sketch. It is not the goal to explain every last detail but the outline should be made clear. It is certainly not required to fully understand every algorithm in order to use it but it's good if you know what you're doing. After the theoretical part the interface description follows. Note the *Type notations*. The final part covers usually some example code that supports the understanding of the interfaces.

CHAPTER

ONE

CONTENTS

1.1 Introduction

Contents

1.1.1 About this library

This library provides several algorithms from the field of machine learning and statistics. A short introduction into statistics is given in the next chapter. The goal of this documentation is to provide a theoretical background additional to the implementation. Main topics covered so far are sampling methods, model fitting and model selection. The model fitting module provides various many different learning algorithms. Where possible, it was tried to stick to a statistics and information theory framework.

Besides the python standard library, the framework mainly uses scipy and numpy. Although not all algorithms rely on them, it's strongly encouraged to use those libraries.

Other projects that might be useful in what you try to achieve can be found at scikit. Especially noteworthy is the package scikit-learn that covers a lot of machine learning algorithms. For statistical computation, there's a nice list of projects as StatPy. Noteworthy are also the SciTools package, the MatPlotLib for plotting and the convex optimization package cvxopt.

If you're looking for more theoretical information, note the used *Resources*. On the web, many topics are well covered by wikipedia and also the StatSoft Statistics textbook might be useful.

Type notations

It's important to understand the type signatures. It may help a lot to understand the structure of the code.

Wherever a common type like float or int (or a type that mimics it) is expected, this is noted using the python syntax. Where a list is expected, also the respective python symbol is used. For exampe, [[float]] would be a matrix of floats whereas (float) is a tuple of floats. Lists and tuples are usually exchangable but it's strongly recommended to use the specified one. The syntax for functions and unspecified types is haskell-like. If the there are no restriction on a type, a character is paced instead. Several types are concatenated using an arrow (->) and the last type in the chain is the return value. Consider the example below:

```
>>> (Num b) :: [a] -> b -> a
```

This signature defines a function that takes a list of an arbitrary type and a second parameter of (possibly) another type and returns a value of the same type as the first parameter's list type. Additionally, if a type is requested to behave like a nummerical type (int or float), this is indicated as above for the type b. Several restrictions are seperated by comma. Besides the nummerical restriction, also Ord (orderd type) may appear. A function with this signature could be as simple as

```
>>> lambda lst,idx: lst[idx]
```

Usually, the type symbols are coherent within all methods of a class and often also with related classes (i.e. classes in the same document).

If no type signature is given, the type is completely arbitrary. However, note that this is usually the case for abstract methods and type restrictions may be found in their concrete implementations.

1.1.2 Statistics formulary

Combinatorics

Draws of objects from an urn. There are n objects and k draws.

	Ordered	Unordered
With replacement	n^k	(n+k-1)!/((n-1)!*k!)
Without replacement	n!/(n-k)!	n!/((n-k)!*k!)

Random variables and probability mass function

Consider an experiment with several possible outcomes where the outcome depends on some underlying random process. A prominent example is a coin toss where the possible results are head and tail. The set of all the experiment's outcomes is the **sample space** $\mathcal{S}$. Depending on the nature of the experiment, it may be either discrete or continous.

Such an experiment can be expressed through a **random variable** X. A random variable can be viewed as a function that assigns a value (often a real number) to any element of the sample space $\mathcal{S}$.

$$X : \mathcal{S} \to \mathcal{A}$$

If the sample space $\mathcal{S}$ is continous, then so is the random variable X. If $\mathcal{S}$ is countable, then X is discrete. Note that for the space of a variable, always the calligraphic sign (e.g. $\mathcal{S}, \mathcal{X}, \mathcal{Y}$) is used.

The **probability mass function** $P : \mathcal{A} \to [0, 1]$ gives the probability of seeing a specific outcome of a random variable. If the random variable is continous, this probability has to be zero, so $P(X = x) = 0$. The probability of seeing any

possible outcome must be one, hence

$$\sum_{x \in \mathcal{S}} P(X = x) = 1$$
$$\int_{x \in \mathcal{S}} P(X = x)dx = 1$$

Usually, the random variable is denoted with an uppercase letter (X) whereas a lower case letter (x) is used for a specific value of the variable. Also, in order to make mathematical terms more readable, the following notation is used:

$$P(x) := P(X = x)$$

Given two random variables X, Y, the probability that the respective outcomes are x and y are observed at the same time is called the **joint probability** $P(x, y) := P(X = x, Y = y)$. Two random variables are **independent**, if and only if

$$P(x, y) = P(x)P(y)$$

holds. Independence means that the outcome of one random variable does not influence the outcome of the other one. If two random variables are independent, one can easily find the probability of one of the two variables by summing or integration over the other one, i.e. $P(x) = \sum_y P(x, y)$ if the variables are discrete.

The **conditional probability** $P(x|y)$ is the probability of seeing a certain outcome x, given that the outcome of the other random variable was already observed to be y.

$$P(x|y) = P(X = x \,|\, Y = y) := \frac{P(x, y)}{P(y)}$$

For independent variables, it can be seen that $P(x|y) = P(x)$, which intuitively makes sense as y doesn't influence x.

The law of **total probability**, also called **marginalization**, states the following:

$$P(x) = \sum_{y \in \mathcal{Y}} P(x|y)P(y)$$
$$P(x) = \int_{y \in \mathcal{Y}} P(x|y)P(y)dy$$

Bayes

The **Bayes theorem** states the following:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}$$

The term $P(x)$ is the prior probability, as its independent of y and can thus be interpreted as the probability of an outcome of X where no information about the second variable Y is available. In this view, the conditional probability $P(x|y)$ is the posterior, so the probability after some information was already observed. The denominator $P(y)$ is for normalization, hence also called normalizer.

Using the law of total probability, the theorem can be re-written into the following form:

$$P(x|y) = \frac{P(y|x)P(x)}{\sum_{x' \in \mathcal{X}} P(y|x')P(x')}$$

If there are three random variables, the probability of the outcome x, given that the two outcomes y, z are observed is

$$P(x|y,z) = \frac{P(y|x,z)P(x|z)}{P(y|z)}$$

Two variables X, Y are **conditionally independent**, iff

$$P(x,y|z) = P(x|z)P(y|z)$$

given a third random variable Z. It follows that

$$P(x|z) = P(x|y,z)$$
$$P(y|z) = P(y|x,z)$$

In robotics, the term **Belief** is sometime used. Let $u_1, z_1, \ldots, u_t, z_t$ be a sequence of measurements z_t and actions u_t up to time t. The belief expressed the probability of being in the current state x_t:

$$\text{Bel}(x_t) = P(x_t \,|\, u_1, z_1, \ldots, u_t, z_t)$$

Probability distributions

The term **probability distribution** is used to describe the collection of the probabilities for all possible outcomes. When talking about a probability distribution, usally the probability mass or density functions are referred to.

Remember the **probability mass function** (pmf) $P(X = x)$, already defined above. It expresses how probable a certain outcome is and it holds that

$$\sum_{xin\mathcal{X}} P(X = x) = 1$$

In the discrete case, also the probability of several outcomes can be computed easily by

$$P(X \in A) = \sum_{x \in A} P(X = x)$$

For discrete $P(X)$, the probability of any element is non-negative and strictly positive for at least one $x \in \mathcal{X}$. On the other hand, if $P(X)$ is continous, the probability of one exact value goes to zero, hence $P(X = x) = 0 \, \forall x$. In consequence, for continous functions we define

$$P(a \leq X \leq b) = \int_a^b p(x)dx$$

with $p(x)$ the **probability density function** (pdf). Analogous to the pmf, the **probability density function** describes how likely a certain outcome is for the random varaible. The pdf is also non-negative all probabilities sums up to one, $\int_{-\infty}^{\infty} p(x)dx = 1$.

Further, we can define the **cumulative distribution function** (cdf):

$$F(x) := P(X \leq x) = \int_{-\infty}^{x} f(t)dt$$

which expresses the probability that the outcome of the experiment will be no larger than x. Note the limits $\lim_{x \to -\infty} F(x) = 0$ and $\lim_{x \to \infty} F(x) = 1$. From this definition, it's easily seen that

$$F(b) - F(a) = P(a \leq X \leq b) = \int_{a}^{b} f(x)dx$$

If $P(X)$ isn't continous at all places x, the probability at the discontinuity is

$$P(X = x) = F(x) - \lim_{y \to x-} F(y)$$

which of course again goes to zero if $P(X)$ is continuous at x.

For sampling, the **inverse cdf** would be important. Unfortunately, a closed form representation is often not available. The inverse cdf is defined as

$$F^{-1}(y) := \inf_{x} F(x) \geq y \qquad y \in [0, 1]$$

In other words this means that for a given y, find the value x such that $F(x) = y$.

There are several well known distributions defined in the section *Distribution Wrappers* and also in the scipy library.

Statistics

Consider a set of data points. Likely, one wants to give some quantitative description of their shape. Also, one would like to be able to give some characterisations of known and well defined probability distributions.

For this, let's first define the **Moment**

$$\mu_k := E[X^k]$$

using the definition of the **Expected value** (for discrete and continous random variables)

$$E[X] = \sum_{x} xP(x)$$

$$E[X] = \int_{x} xf(x)dx$$

Note that $f(x)$ is the probability density function of X. The expected value is sometimes referenced as mean and misleadingly associated with the arithmetic mean. Of course this is only true, if the probability of the observations is uniform (all observations are equally likely) which is often the case for measured data (then the probability distribution is expressed by repetition of measurements).

The expected value is linear, a fact that is formally noted by the following equation:

$$E[aX + bY + c] = a\,E[X] + b\,E[Y] + c$$

If the two random variables X, Y are independent, it also holds that

$$E[X\,Y] = E[X]\,E[Y]$$

The **Central Moment** is the moment about the expected value, thus

$$\mu_k := E[(X - E[X])^k]$$

From its definition, one can easily see that the zero'th central moment is one and the first central moment is zero.

$$\mu_0 = 1$$
$$\mu_1 = 0$$

The second central moment is the **Variance**

$$\begin{aligned} \text{Var}[X] &:= E[(X - E[X])^2] \\ &= \sum_x (x - \mu)^2 P(x) \\ &= \int_x (x - \mu)^2 f(x) dx \end{aligned}$$

again $f(x)$ is the probability density function and $\mu = E[X]$ the expected value. The equations show the definition for a discrete and continous random variable, respectively. An alternative and often more appropriate formulation is

$$\text{Var}[X] = E[X^2] - (E[X])^2$$

The variance is pseudo-linear, which means the following:

$$\text{Var}[aX + b] = a^2\text{Var}[X]$$

A generalisation of the variance to multiple random variables yields the **Covariance**, which is defined analogously

$$\text{Cov}[X, Y] = E[(X - E[X])\,(Y - E[Y])]$$

For the sake of completeness but without further comment, here are some properties of the covariance:

$$\begin{aligned} \text{Cov}[X, X] &= \text{Var}[X] \\ \text{Cov}[X, Y] &= E[X\,Y] - E[X]\,E[Y] \\ \text{Cov}[X, Y] &= \text{Cov}[Y, X] \\ \text{Cov}[aX + b, Y] &= a\text{Cov}[X, Y] \\ \text{Cov}[X + Y, Z] &= \text{Cov}[X, Z] + \text{Cov}[Y, Z] \end{aligned}$$

1.2 core — Core functionality

Contents

1.2.1 Abbreviations

Some list abbreviations:

```
head = lambda lst: lst[0]
tail = lambda lst: lst[1:]
last = lambda lst: lst[-1]
init = lambda lst: lst[:-1]
```

And for tuples:

```
fst = lambda lst: lst[0]
snd = lambda lst: lst[1]
```

1.2.2 Basic math

sign (x)

Return -1 if x is smaller than zero. If x is equal to zero, return zero. Return +1 otherwise.

This method implements the sign function, which is defined as

$$\text{sign}(x) = \begin{cases} -1 & , x < 0 \\ 0 & , x = 0 \\ +1 & , x > 0 \end{cases}$$

Note that the abbreviation **sgn** is also available:

```
sgn := sign
```

prod (*lst*)
Return the product of all list elements.

$$\text{prod}(x) = \prod_{i=1}^{N} x_i = x_1 x_2 \dots x_n$$

span (*lst*)
Return the span of a list.

$$\text{span}(x) = \max(x) - \min(x)$$

argmin (*f, arg*)
Return the argument for which the value of a function is minimal.

The function *f* is evaluated at each element of the argument list *arg*. The value in *arg* which returns the lowest value of *f* is returned.

Parameters

- *f* ((Ord b) => a -> b) – Function to be evaluated.
- *arg* (a) – List of argument values.

```
>>> l1, l2, f = [4,5,6], [1,2,3], lambda x,y: x**2-y**2
>>> argmin(lambda (x,y): f(x,y), itertools.product(l1, l2))
(4, 3)
```

$$\arg\min_{x \in \mathcal{X}} f(x)$$

argmax (*f, arg*)
Return the argument for which the value of a function is maximal.

The function *f* is evaluated at each element of the argument list *arg*. The value in *arg* which returns the lowest value of *f* is returned.

Parameters

- *f* ((Ord b) => a -> b) – Function to be evaluated.
- *arg* (a) – List of argument values.

```
>>> l1, l2, f = [4,5,6], [1,2,3], lambda x,y: x**2-y**2
>>> argmax(lambda (x,y): f(x,y), itertools.product(l1, l2))
(6, 1)
```

$$\arg\max_{x \in \mathcal{X}} f(x)$$

normalize (*lst, s=1.0*)
Return the list, such that the elements sum up to *s*.

Parameters

- *lst* – A list of numerals.
- *s* – Normalize scale.

Returns Scaled list with sum(*lst*) = *nrm*

```
>>> normalize([2.0,4.0, 6.0], 3.0)
[0.5, 1.0, 1.5]

>>> sum(normalize([2.0,4.0, 6.0], 3.0))
3.0
```

$$\vec{x}_n = s\frac{\vec{x}}{\|x\|}$$

1.2.3 Distance measurements

Kullback-Leibler Divergence

The Kullback-Leibler Divergence is defined as:

$$D_{\mathrm{KL}}(p||q) = \int_{-\infty}^{\infty} p(x)\log_2\frac{p(x)}{q(x)}dx.$$

KLD (*p, q, r, save=True*)
Return the continous Kullback-Leibler divergence.

q(x) must be non-zero for all x in *r*.* If this doesn't hold for *q*, then *inf* is returned. If the *save* flag is set, the support *r* is reduced to the support of *q* (thus *q* non-zero for all x in *r'*).

If you have lists instead of functions, use `listFunc()` and `range()`.

Parameters

- *p* ((Num b) :: a -> b) – pdf of the first distribution.
- *q* ((Num b) :: a -> b) – pdf of the second distribution.
- *r* ([a]) – Support.
- *save* (bool) – Flag, if support may be reduced.

Returns KLD of *p* and *q*, measured at locations in *r*.

```
>>> KLD(lambda x: x, lambda x: x-1, [0,1,2,3,4], save=True)
5.4150374992788439

>>> KLD(lambda x: x, lambda x: x-1, [0,1,2,3,4], save=True)
inf
```

Time warping

Time warping is a distance measurement for two vectors of different size.

LTW (*x, y, dist=<function euclidean at 0x3319500>*)
Linear time warping.

Parameters

- *x* ([a]) – First input vector.
- *y* ([b]) – Second input vector.
- *dist* ((Num c) => a -> b -> c) – Distance measurement for vector elements.

Returns Linear time warping distance of *x*, *y*.

```
>>> LTW([1,2,3,4,5,6,7], [9,8,7,6])
28.0
```

Given two input vectors $\vec{x}, \vec{y}$ with lenghts I, J.

$$w(i) = \text{Int}\left[\frac{J-1}{I-1}(i-1) + 1 + 0.5\right]$$

$$D(\vec{x}, \vec{y} = \sum_{i=1}^{I} d(x_i, y_{w(i)})$$

1.2.4 Statistical basics

Average

For all average measurements, the argument *lst* must be a list of numerics. Note that functions don't actually need to be implemented as presented here.

mean (*lst*)
Return the arithmetic mean of all values of the given list.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

median (*lst*)
Return the median value of the given list.

gMean (*lst*)
Return the geometric mean of the given values.

$$M_g(x) = \sqrt[N]{\prod_{i=1}^{N} x_i}$$

hMean (*lst*)
Return the harmonic mean of the given values.

$$M_h(x) = \frac{N}{\sum_{i=1}^{N} \frac{1}{x_i}}$$

qMean (*lst*)
Return the quadratic mean of the given values.

$$M_2(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

cMean (*lst*)
Return the cubic mean of the given values.

$$M_3(x) = \sqrt[3]{\frac{1}{N}\sum_{i=1}^{N} x_i^3}$$

genMean (*lst, p*)
Return the generalized mean of the given values.

The generalized mean with exponent *p* is defined as follows:

$$M_p(x) = \sqrt[p]{\frac{1}{N}\sum_{i=1}^{N} x_i^p}$$

From this, one can clearly see the following equivalences:

```
mean = genMean(lst, 1.0)
qMean = genMean(lst, 2.0)
cMean = genMean(lst, 3.0)
```

rms (*lst*)
Return the root mean square of the given values.

$$\mathrm{RMS}(x) = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

```
>>> rms = qMean
```

midrange (*lst*)
Return the midrange of the given values.

$$\mathrm{MID}(x) = \frac{\max(x) + \min(x)}{2}$$

var (*lst*)
Return the variance of the given list.

Generally, the variance is defined as:

$$\mathrm{Var}(x) = E[(X - \mu)^2] = E[X^2] - (E[X])^2$$

Here, it is implemented using the arithmetic mean:

$$\mathrm{Var}(x) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

rss (*lst*)
Return the residual sum of squares.

The list items are expected to be of the form (f(x),y).

$$\mathrm{RSS} = \sum_{i=1}^{N}(y_i - f(x_i))^2$$

binning (*data, numBins, hist=False*)
Divide a list into multiple bins of equal intervalls.

For the type a there must be a float representation and it must be ordered. Specifically, the type must support the functions *min*, *max* and *float*.

Parameters

- *data* ([a]) – Input list.
- *numBins* (int) – Number of bins.
- *hist* (bool) – Return the number of samples per bin instead.

Returns A list of bins, containing the data elements.

```
>>> binning([1,2,3,4], 2, hist=False)
[[1,2],[3,4]]
```

binRange (*data, numBins*)
Return the intervall boundaries used by *binning*.

The returned intervalls describe the lower and upper limit, whereas the lower limit is inclusive and the upper limit is inclusive only for the last intervall.

Parameters

- *data* (a) – Input list.
- *numBins* (int) – Number of bins.

Returns List of intervalls.

```
>>> binRange([1,2,3,4], 2, hist=False)
[(1.0,2.5),(2.5,4.0)]
```

histogram(*data*)
Count the number of occurrences of tokens in a list.

Parameter *data* – List of tokens.

Returns Dict with token as key and relative frequency as value

majorityVote(*data, tieBreaking=<function <lambda> at 0x331da28>*)
Return the token with the highest number of occurrences.

The tie-breaking rule is applied, if there are several tokens with identical number of occurrences. The *tieBreaking* function takes two arguments: The tokens with maximal votes and the token histogram. It has to select a single token as a winner. The default function is a random choice.

Parameters

- *data* (a) – Stream of input tokens.
- *tieBreaking* ([a] -> {a -> Int} -> a) – Function to determine the vote winner, in the case of a tie.

Returns Token of the data stream with the highest number of occurrences.

```
>>> majorityVote([1,2,1], tieBreaking=lambda l,h: max(l))
1


>>> majorityVote([1,2,1,2,3], tieBreaking=lambda l,h: max(l))
2
```

1.2.5 Distribution Wrappers

So far, no wrapper is fully implemented. Use the distribution related methods of scipy.stats.

1.2.6 Data conversion

isList(*l*)
Check, if l is a list type or not.

listFunc(*lst, off=0*)
Represent a list *lst* through a function f, such that f(x) = *lst* [x + *off*].

1.3 sampling — Sampling Methods

Contents

Often, one would like to accquire some samples that follow a certain probability distribution. For example, in algorithm testing, training samples are required but real-world measurements are too rare to be of use, so artificial samples have to be generated. In almost any case, the random samples have to fulfill some constraints, expressed through the shape of how the samples are distributed. Thus, the standard scenario is that a probability distribution is given (i.e. assumed) and one requires samples that - in high numbers - match the prerequisited distribution.

1.3.1 Sampling from known distributions

In the most simple case, a well-known *distribution function* is provided. A probability distribution can be characterized by the probability density function $p(x)$

$$P(a \leq X \leq b) = \int_a^b p(x)dx$$

and the cumulative distribution function $F(x)$

$$F(x) := P(X \leq x) = \int_{-\infty}^{x} p(t)dt$$

given the *probability mass function* $P(X)$. For sampling, the inverse cumulative distribution function $F^{-1}(y)$ (**i-cdf**) is especially interesting. If a closed form representation exists, one can sample the uniform distribution (which usually is relatively easy) in the i-cdf's domain $y \in [0, 1]$ and then use the i-cdf to retrieve samples distributed according to its probability distribution:

1. Draw $u \sim \mathcal{U}^{[0,1]}$
2. Obtain a sample $s = F^{-1}(u)$

The numpy.random package contains several functions to sample directly from the most prominent distributions. Unfortunately, the inverse cdf does not always exist, which makes more complex methods inevitable.

1.3.2 Sampling from arbitrary distributions

The target sample distribution may not always be known (e.g. it can be determined by measurements from a source with unknown distribution) or a closed form representation of the i-cdf may could not be available. In this case, a more elaborate mechanism has to be made use of (which, as always, comes with the cost of less efficient computation).

Rejection sampling

This is a relatively simple but often inefficient sampling scheme. A proposal distribution $q(x)$ is required as well as a scaling constant $M < \infty$. The proposal distribution can have any shape, but it must be guaranteed that

$$Mq(x) \geq p(x) \quad \forall x \in \mathcal{X}$$

with $p(x)$ the target distribution. To find any tuple $(q(x), M)$ that fullfills the above equation is often not very hard find. However, as described later, the larger the scaling, the less efficient the algorithm becomes and finding an acceptable tuple may be difficult.

The algorithm can then be laid out:

1. Iterate, until N accepted samples have been generated
   (a) Sample from the proposal $x \sim Q$
   (b) Sample from the uniform distribution $u \sim U^{[0,1]}$
   (c) Accept the sample x, if $u < \frac{p(x)}{Mq(x)}$. Reject it otherwise.

As it can be seen from the algorithm, a sample has to be obtained from the distribution $q(x)$, hence a proposal distribution should be chosen that is relatively easy to sample. A sample is accepted, iff $u < \frac{p(x)}{Mq(x)}$, thus the probability that a sample is accepted is proportional to $\frac{1}{M}$. If the proposal and target distributions don't match well, the scaling has to be large which introduces massive performance issues.

1.3.3 Interfaces

`rouletteWheelN` (*data, weights, numSamples*)
Return samples drawn from a discrete distribution.

Parameters

- *data* ([a]) – List of tokens.
- *weights* ([float]) – Weights of tokens.
- *numSamples* (int) – Number of samples to draw.

Returns *numSamples* samples drawn from *data* according to their weights.

`rouletteWheel` (*data, weights*)
Return a sample drawn from a discrete distribution.

Parameters

- *data* ([a]) – List of tokens.
- *weights* ([float]) – Weights of tokens.

Returns Generator object for samples drawn from *data* according to their weights.

`rejectionSamplerN` (*qSampler, qDist, pDist, N, M*)
Draw samples from a distribution, given its probability density function.

This sampler can be used, if the probability density function of the target distribution is known, but there's no direct sampling approach (e.g. the inverse cdf is not known).

Parameters

- *qSampler* (a) – Sampling function of the proposal distribution.
- *qDist* ((Num b) => a -> b) – Probability density function of the proposal distribution.
- *pDist* ((Num b) => a -> b) – Probability density function of the target distribution.
- *N* (int) – Number of samples.
- *M* (float) – Scale parameter.

Returns N samples, distributed according to the probability density of the target distribution *pDist*.

`rejectionSampler` (*qSampler, qDist, pDist, M*)
Draw samples from a distribution, given its probability density function.

This sampler can be used, if the probability density function of the target distribution is known, but there's no direct sampling approach (e.g. the inverse cdf is not known).

Parameters

- *qSampler* (a) – Sampling function of the proposal distribution.
- *qDist* ((Num b) => a -> b) – Probability density function of the proposal distribution.
- *pDist* ((Num b) => a -> b) – Probability density function of the target distribution.
- *M* (float) – Scale parameter.

Returns Generator for samples, distributed according to the probability density of the target distribution *pDist*.

metropolisHastingsSampler (*qSampler, qDist, pDist, x0=0.0*)

Draw samples from a distribution.

Parameters

- *qSampler* (a -> a) – Sampling function of the proposal distribution.
- *qDist* ((Num b) => a -> a -> b) – Probability density function of the proposal distribution.
- *pDist* ((Num b) => a -> b) – Probability density function of the target distribution.
- *x0* (a) – Initial value

Returns Generator object for samples, distributed according to the target distribution *pDist*.

The proposal is

$$\min\left(1, \frac{p(x^*)q(x_{t-1}, x^*)}{p(x_{t-1})q(x^*, x_{t-1})}\right)$$

metropolisSampler (*qSampler, qDist, pDist, x0=0.0*)

Draw samples from a distribution.

Parameters

- *qSampler* (a -> a) – Sampling function of the proposal distribution.
- *pDist* ((Num b) => a -> b) – Probability density function of the target distribution.
- *x0* (a) – Initial value

Returns *numSamples* samples, distributed according to the target distribution *pDist*.

The metropolis sampler assumes a symmetric proposal, i.e.

$$q(x^*, x_t) = q(x_t, x^*)$$

This reduces the proposal to

$$\min\left(1, \frac{p(x^*)}{p(x_{t-1})}\right)$$

1.3.4 Examples

1.4 `fitting` — Model Fitting

Contents

1.4.1 Introduction

The term model is used in a very general manner. A model usually is a mathematical (often a statistical) tool that can be adjusted to some given data points. Once set up (trained), it can be evaluated at arbitrary input values. There are roughly two types of models:

1. Classification
2. Regression

In classification problems an input point has to be assigned to a cluster. If the clusters are labeled, the collection of all clusters defines a codebook, so that the problem can also be viewed as finding the optimal code for each input value. Loosely speaking regression means curve fitting, i.e. adjust parameters of a mathematical function such that it optimally represents the training data. Of course, it has to be defined what optimal means. There is no general answer to this question, as it depends on the fitting target (the model). Often, the residual sum of squares is used, i.e. the sum of squared deviation between measurement and prediction from the model.

To be clear, the model only specifies how training data is explained. For learning problems, the model is incomplete (some information is missing, e.g. parameters) and there's need for a learning algorithm that determines this information. The model learning algorithms strongly depend on the input their given. Basically, there are two situations:

1. Supervised learning
2. Unsupervised learning

For supervised learning, the input values are measured together with the target output values. Both are presented to the fitting algorithm, so that it can train the model to fit the targets optimally. However, it's not always the case that targets are available. If there are only input but no output values, it's an unsupervised learning problem. This implies that it's not possible to use an error measurement for training or evaluation. So unsupervised learning algorithms try to find some (hidden) structure in the input data. A typical example is finding the location where the density of input points is maximal.

1.4.2 Data model

There are three kinds of information:

1. Features
2. Labels
3. Weights

A feature is the same as one input value. It may be a single value, however the inputs are often multidimensional, so in general the features are vectors (lists). The labels are the target values to the respective input. In most cases, the labels are unidimensional. However, the concrete data format (of input and output values) strongly depends on the model and thus is specified by the learning algorithm. Some algorithms also allow weighted samples. The weights are usually real numbered values, passed as a list to the learning algorithm.

Usually, the training data is passed to the `fit` method seperately through the respective parameters. For unsupervised learning, the *labels* argument will be ignored. Internally, there are several ways to represent training data. For details, see *Data conversion*.

In this library, the model and learning algorithm are combined in a single class. The class `Model` defines the interface for all learning algorithms (supervised or unsupervised).

class Model()
Interface for fitting algorithm.

err(*(x, y)*)
Return the distance between the target *y* and the model prediction at the point *x*.

eval(*x*)
Evaluate the model at data point *x*.

fit(*features, labels=None, weights=None*)
Fit the model to the training data.
Parameters
- *features* – Training samples.
- *labels* – Target values.
- *weights* ([float]) – Sample weights.

Returns self

Many learning algorithms use several models to make the prediction more robust. For this, a very general interface is defined. The class `Committee` allows to collect models and it provides different mixins for evaluation and error measurement.

class Committee()
Bases: `ailib.fitting.model.Model`

Interface for fitting algorithms that base on an ensemble of several models.

Typically, the committee consists of several — possibly different — submodels. All submodels are trained first, where the manner in which this happens is not known here (thus the *fit* method is not overwritten). For prediction, each trained submodel is evaluated. The final result is then based on the individual predictions. How the predictions are assembled depends on the problem type.

The mixin classes provide common evaluation and error functions. Use like so:

```
>>> class foo(Committee.Sampling, Committee): pass
```

class Classification()
Mixin for classification models.

err(*(x, y)*)
0-1 loss function for classification.

eval(*x*)
Return the most frequently predicted class of *x*.

class Regression()
Mixin for regression models.

err(*(x, y)*)
Squared residual.

eval(*x*)
Return the average model prediction at *x*.

class Sampling()
Mixin for classification models. The class label is sampled from all predicted labels.

err (*(x, y)*)
0-1 loss function for classification.

eval (*x*)
Return the class of *x*, sampled from the predictions of the individual models.

addModel (*m*)
Add a model *m* to the committee.

1.4.3 Learning algorithms

Supervised learning

Linear Regression

Contents

Generalized linear models

Given an input vector $\vec{x} = [x_1, x_2, \ldots, x_p]^T$ and coefficients $\vec{\beta} = [\beta_0, \beta_1, \beta_2, \ldots, \beta_p]^T$. A generalized linear model is the function of the form:

$$f(\vec{x}) = \beta_0 + \sum_{j=1}^{p} x_j \beta_j$$

This notation describes any function with linear dependence on the parameters β_j. For example, a polynomial can be written as

$$f(\vec{x}) = \beta_0 + \sum_{j=1}^{p} x_j \beta_j \quad \text{with } \vec{x} = \left[x, x^2, \ldots, x^p\right]$$

The fitting problem focuses on finding the *optimal* parameters β_j of the function $f(\vec{x})$ for N given training pairs $(\vec{x}_i, y_i)$, with $y_i = f(\vec{x}_i)$ being the target output. In this context, optimal means that the allocation of $\vec{\beta}$ needs to minimize the sum of squared residuals:

$$\begin{aligned} \text{argmin}_\beta \, \text{RSS}(\beta) &= \text{argmin}_\beta \sum_{i=1}^{N} w_i \left(y_i - f(x_i)\right)^2 \\ &= \text{argmin}_\beta \left(\vec{y} - \mathbf{X}\vec{\beta}\right)^T W \left(\vec{y} - \mathbf{X}\vec{\beta}\right) \end{aligned}$$

The lower equation is just there to get rid of the sum. The matrix $\mathbf{X}$ holds all the training input vectors (size Nx$(p+1)$), the matrix W is a diagonal matrix with $W_{ii} = w_i$.

As known from basic analysis, the extremal points (here the minimum) can be found by setting the function derivative to zero. In the problem setting, the training samples are fixed and the optimization goes for the parameters, thus we're interested in the derivative w.r.t the parameters β

$$\frac{\partial \text{RSS}(\beta)}{\partial \beta} = 0$$

The nice property of generalized linear models is that this derivation can easily be determined:

$$\frac{\partial \text{RSS}(\beta)}{\partial \beta} = -2\mathbf{X}^T\mathbf{W}\left(\vec{y} - \mathbf{X}\vec{\beta}\right)$$

Putting the above ideas toghether yields the closed form solution of the fitting problem:

$$\begin{aligned}
\frac{\partial \text{RSS}(\beta)}{\partial \beta} &= 0 \\
\mathbf{X}^T\mathbf{W}\left(\vec{y} - \mathbf{X}\vec{\beta}\right) &= 0 \\
\mathbf{X}^T\mathbf{W}\vec{y} - \mathbf{X}^T\mathbf{W}\mathbf{X}\vec{\beta} &= 0 \\
\mathbf{X}^T\mathbf{W}\vec{y} &= \mathbf{X}^T\mathbf{W}\mathbf{X}\vec{\beta} \\
\left(\mathbf{X}^T\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{W}\vec{y} &= \vec{\beta}
\end{aligned}$$

The generalization to K outputs can be done, if $\vec{y}$ is written as (NxK)-matrix with each row a set of output values. Then the coefficients are computed the same way as above, $\vec{\beta}$ becoming a $((p+1)\text{x}K)$ matrix. Note that the outputs are independent of each other, thus solving the system with multiple outputs it is equivalent to solving the system once for each single output. [ESLII]

Ridge regression The above described least squares routine may lead to severe *overfitting* problems. The *generalization error* might even decrease, if some of the coefficients are set to zero.

Instead of removing a coefficient subset, the ridge regression introduces a penalty on the size of the coefficients. The idea is that the sum of squared coefficients may not exceed a threshold. This forces less significant coefficients to vanish while the most influencing ones shouldn't be changed too heavily. Using lagrange multipliers, the optimization problem is stated the following way:

$$\hat{\beta}^{\text{ridge}} = \text{argmin}_\beta\, w_i \left(\sum_{i=1}^{N} (y_i - f_\beta(x_i))^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right)$$

The parameter $\lambda \geq 0$ is introduced to control the amount of shrinkage. The larger λ, the more the coefficients are constrained. For $\lambda = 0$, the problem becomes identical to the *ordinary least squares*.

As for ordinary least squares, the sum of squared residuals can be formulated

$$\text{RSS}(\lambda) = (\vec{y} - \mathbf{X}\vec{\beta})^T\mathbf{W}(\vec{y} - \mathbf{X}\vec{\beta}) + \lambda\vec{\beta}^T\vec{\beta}$$

and by its derivation with respect to β, the closed form solution is found:

$$\hat{\beta}^{\text{ridge}} = \left(\mathbf{X}^T\mathbf{W}\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{W}\vec{y}$$

Interfaces

class `LinearModel` (*lambda_=0.0*)
Generalized linear model base class.

Although possible, it's generally not a good idea to use this class directly. Instead, use a derived classes, such as `BasisRegression`.

Subclasses are required to write the *coeff* member after fitting. Otherwise some routine calls may fail or produce wrong results.

Parameter *lambda* (float) – Shrinkage factor.

`err` (*x, y*)
Squared residual.

`eval` (*x*)
Evaluate the model at data point *x*.

The last computed coefficients are used.

Todo

This routine doesn't work, if *x* is a scipy.matrix

Parameter *x* ([float]) – (p+1) Basis inputs.

Returns Model output.

`fit` (*data, labels, weights=None*)
Fit the model to some training data.

Parameters

- *data* ([a]) – Training inputs.
- *labels* ([b]) – Training labels.
- *weights* ([float]) – Sample weights.

Returns self

`numCoeff` ( )
Number of basis values.

`numOutput` ( )
Number of output values.

class `BasisRegression` (*lambda_=0.0*)
Least squares regression with arbitrary basis input.

This class allows to use arbitrary basis functions. The resulting model is

f(x) = c0*b0(x) + c1*b1(x) + c2*b2(x) + c3*b3(x) + ...

with b0(x) := 1.0

The basis functions can be registered using the *addBasis* method. Any function can be used, the only requirement is that the type of its only argument is the same type as the model input *x* (see below).

Parameter *lambda* (float) – Shrinkage factor.

`addBasis` (*b*)
Add a basis function.

Parameter *b* ((Num b) :: a -> b) – Basis function.

eval (*x*)
Evaluate the model at data point *x*.

fit (*data, labels, weights=None*)
Fit the model to some training data.

Parameters

- *data* (a) – Training inputs.
- *labels* (b) – Training labels.
- *weights* ([float]) – Sample weights.

Returns self

class PolynomialRegression (*deg, lambda_=0.0*)
Polynomial regression.

For fitting a polynomial f(x) = c0 + c1*x + c2*x**2 + c3*x**3 + ...

Parameters

- *deg* (int) – Degree of polynomial (largest exponent)
- *lambda* (float) – Shrinkage factor.

Examples

Let's define a toy function that generates some samples and fits a polynom using them.

```
def polyEst(lo, hi, numSamples, sigma, deg, l=0.0):

    # Setting up the true polynomial and estimator
    poly = lambda x: 1.0 + 2.0*x  + 3.0*x**2 + 0.5*x**3 + 0.25*x**4 + 0.125*x**5
    est = ailib.fitting.PolynomialRegression(deg, lambda_=l)

    # Generate samples from the polynomial
    x = [lo + (hi - lo) * random.random() for i in range(numSamples)]
    y = [poly(i) + random.gauss(0.0, sigma) for i in x]

    # Fit the estimator to the samples
    est.fit(x,y)
    return est, (x,y)

# y = exp(a0*x0) * exp(a1*x1) * exp(a2*x2)
# ln(y) = ln( ... ) = ln(exp(a0*x0)) + ln(exp(a1*x1) + ln(exp(a2*x2)) = a0*x0 + a1*x1 + a2*x2


>>> print est(0.0, 1.0, 1000, 1.0, 5)
Poly(1.06417448201 * x**0 + 1.90233085087 * x**1 + -1.80439247299 * x**2 + 25.2719255838 * x**3 + -38
```

Non-Linear least squares

Contents

Where not otherwise denoted, the main sources are [IMM3215] and wikipedia.

Let $M_\theta(x)$ be a function of x with parameters θ. Unlike *Generalized linear models*, the function M may be nonlinear in the parameters θ, for example

$$M_\theta(x) = \theta_0 \exp(\theta_1 x) + \theta_2 \exp(\theta_3 x) \quad \text{with } \theta = [\theta_0, \theta_1, \theta_2, \theta_3]^T$$

The aim of non-linear least squares is to find the parameters θ such that the function fits a set of training data (input and output values of M) optimally with respect to the sum of squares residual.

$$\theta^* = \text{argmin}_\theta \sum_i (y_i - M_\theta(x_i))^2$$

The input and output values x, y can be considered to be constant, as the minimization is over the function parameters θ. To make the notation simpler, the functions f, F are further defined as:

$$\begin{aligned} f_i(\theta) &= y_i - M_\theta(x_i) \\ F(\theta) &= \sum_i (y_i - M_\theta(x_i))^2 = \sum_i f_i(\theta)^2 \end{aligned}$$

and the optimization problem can therefore be reformulated:

$$\theta^* = \text{argmin}_\theta F(\theta)$$

Note: In contrast to *Generalized linear models*, the presented algorithms only find local minimal. A local minima of a function F is defined by the following equation, given a (small) region δ:

$$F_(\theta^*) \leq F_(\theta) \quad \forall \|\theta - \theta^*\| < \delta$$

Gradient descent methods

Descent methods iteratively compute the set θ_{t+1} from a prior set θ_t by

$$\theta_{t+1} = \theta_t + \alpha \vec{h}$$

with the initial set θ_0 given. The moving direction $\vec{h}$ and step size α have to be determined in such a way that the step was actually a descent, i.e. the following equation holds for each step t:

$$F(\theta_{t+1}) < F(\theta_t)$$

The step size α can be set to a constant, e.g. $\alpha = 1$ or determined by *Line search* (or another method). A constant value is justified as the line search algorithm may take a lot of time.

Steepest descent Let $\nabla(\theta)$ be the gradient of $F(\theta)$.

$$\nabla(\theta) := \left[\frac{\partial F(\theta)}{\partial \theta_0}, \frac{\partial F(\theta)}{\partial \theta_1}, \ldots, \frac{\partial F(\theta)}{\partial \theta_n}\right]^T$$

The gradient $\nabla(\theta)$ can be seen as the direction of the steepest increase in the function $F(\theta)$. Hence, the **steepest descent method** or also named **gradient method** uses its negative as moving direction $\vec{h} = -\nabla(\theta)$.

The choice of the gradient is reasonable, as it is locally the best choice since the descent is maximal. This method converges but convergence is only linear, thus it is often used if θ is still far from the minimum θ^*.

The steepest descent algorithm can be summarized as follows:

1. Init $\theta = \theta_0$.
2. **Iterate**
   (a) $\vec{h} = -\nabla(\theta)$.
   (b) Compute α (using *Line search* or another method).
   (c) $\theta = \theta + \alpha\vec{h}$.
   (d) Stop, if $\vec{h}$ is small enough.

Newton's method Let $H(\theta)$ be the Hessian matrix of $F(\theta)$

$$\mathbf{H}(\theta) = \left[\frac{\partial^2 F(\theta)}{\partial \theta_i \, \partial \theta_j}\right]$$

As θ^* is an extremal point, the gradient must be zero $\nabla(\theta^*) = 0$. Thus, θ^* is a stationary point of the gradient. Using a second order Taylor expansion of the gradient around θ, one gets the equation

$$\nabla(\theta + \vec{h}) \; = \nabla(\theta) + \mathbf{H}(\theta)\vec{h}$$

Setting $\nabla(\theta + \vec{h}) = 0$ yields the moving direction of **Newton's method**:

$$\mathbf{H}(\theta)\vec{h} = -\nabla(\theta)$$

The algorithm can be defined analogous to the *Steepest descent* algorithm.

1. Init $\theta = \theta_0$.
2. **Iterate**

(a) Solve $\mathbf{H}(\theta)\vec{h} = -\nabla(\theta)$.

(b) Compute α (using *Line search* or another method).

(c) $\theta = \theta + \alpha\vec{h}$.

(d) Stop, if $\vec{h}$ is small enough.

Newton's method has up to quadratic convergence but may converge to a maximum. Thus it is only appropriate, if the estimate θ is already close to the minimizer θ^*. A big problem with Newton's method is that the second derivative is required, a prerequisite that can often not be fulfilled.

To combine the strengths of both methods, a **hybrid algorithm** can be defined

1. Init $\theta = \theta_0$.
2. **Iterate**

(a) If $H(\theta)$ is positive definite, Newton's method is used: $\vec{h} = -\mathbf{H}^{-1}(\theta)\nabla(\theta)$.

(b) Otherwise, the gradient method is applied: $\vec{h} = -\nabla(\theta)$.

(c) Compute α (using *Line search* or another method).

(d) $\theta = \theta + \alpha\vec{h}$.

(e) Stop, if $\vec{h}$ is small enough.

Line search All above algorithms require a computation of the step size α. The line search algorithm either computes the optimal or an estimate step size when the current estimate θ and moving direction $\vec{h}$ are given.

Let's consider the function

$$\varphi(\alpha) = F(\theta + \alpha\vec{h})$$

The goal is to find the step size that minimizes the above function. If α is too low, it should be increased in order to speed up convergence. If it is to high, the error might not decrease (if $\varphi(\alpha)$ has a minimum), thus has to be decreased.

A nice property of this function is that its derivation can easily be seen to be

$$\varphi'(\alpha) = \vec{h}^T\nabla(\theta + \alpha\vec{h})$$

The **exact line search** algorithm aims to find a good approximation of the minimum whereas the **soft line search** method only ensures that the $alpha$ is within some boundary. The soft line search algorithm is often preferred, as it is usually faster and a close approximation to the minimum is not required.

The soft line search uses the two following constraints for α_s:

$$\begin{aligned} \varphi(\alpha_s) &\leq \varphi(0) + \varrho\alpha_s\varphi'(0) \\ \varphi'(\alpha_s) &\geq \beta\varphi'(0) \end{aligned}$$

with $\varrho \in (0, 0.25)$ and $\beta \in (\varrho, 1)$. The first constraint ensures that the error decreases, the second one inhibits that the decrease is too small.

The line search algorithm comes from the idea to construct an interval $[a, b]$ which defines the range of acceptable values for α and successively narrow down this interval.

Let's define $\lambda(\alpha) := \varphi(0) + \varrho\alpha\varphi'(0)$, the right hand side of the first constraint. α_{max} is a parameter that inhibits the right side of the interval b to become arbitrary large, in the case of F being unbounded.

1. Initialize $a = 0$

2. Initialize $b = \min(1, \alpha_{\text{max}})$
3. **Iterate while** $\varphi(b) \leq \lambda(b)$ **and** $\varphi'(b) \leq \beta\varphi'(0)$ **and** $b < \alpha_{\textbf{max}}$
    (a) $a = b$
    (b) $b = \min(2b, \alpha_{\text{max}})$
4. $\alpha = b$
5. **Iterate while** $\varphi(\alpha) > \lambda(\alpha)$ **or** $\varphi'(\alpha) < \beta\phi'(0)$
    (a) Refine α, a, b

The first iteration shifts the initial interval to the right as long as the second constraint is not fulfilled. The second iteration shrinks the interval until a solution for α is found for which both constraints hold. The refinement routine is defined as:

1. Initialize $D = b - a$
2. Initialize $c = \frac{\varphi(b) - \varphi(a) - D\varphi'(a)}{D^2}$
3. If $c > 0$, set $\alpha = \min\left(\max\left(a - \frac{\varphi'(a)}{(2c)}, a + 0.1D\right), b - 0.1D\right)$
4. Otherwise, set $\alpha = \frac{a+b}{2}$
5. If $\varphi(\alpha) < \lambda(\alpha)$, set $a = \alpha$
6. Otherwise, set $b = \alpha$

The first branching adjusts the α by approximating φ with a second order polynomial. If the approximation has a minimum, it is taken, otherwise α is set to the midpoint of the interval $[a, b]$. In the first case, it is made sure that the interval decreases by at least 10%. The second branching reduces the interval, such that the new interval still contains acceptable points.

The difference between soft line search and exact line search is the second iteration in the algorithm. For the exact line search, line (5) becomes

1. **Iterate while** $|\varphi'(\alpha)| > \tau|\varphi'(0)|$ **and** $b - a > \epsilon$
    (a) Refine α, a, b

with $\tau, \epsilon > 0$ being error tolerance parameters. [IMM3217]

Gauss-Newton algorithm

Let N be the number of samples and M the number of parameters. Note that $\mathbf{f}(\theta) := [f_0(\theta), f_1(\theta), \ldots, f_{n-1}(\theta)]^T$ is defined. Furthermore, the NxM Jacobian matrix is

$$\mathbf{J}(\theta)_{ij} = \left[\frac{\partial f_i(\theta)}{\partial \theta_j} \right]$$

The **Gauss-Newton algorithm** approximates the function $\mathbf{f}(\theta + \vec{h})$ using a second order taylor expansion $\mathbf{l}(\vec{h})$ around θ. This then is used to formulate the minimization problem $\text{argmin}_{\vec{h}} \tilde{F}(\theta + \vec{h}) := \mathbf{l}^T(\vec{h})\mathbf{l}(\vec{h})$ which is equivalent to solving the linear equation $\left(\mathbf{J}(\theta)^T\mathbf{J}(\theta)\right)\vec{h} = -\mathbf{J}(\theta)^T\mathbf{f}(\theta)$. As in *Newton's method*, the rationale behind this is that the minimum of the approximation can actually be computed. While Newton's method minimizes the approximated gradient ∇, the Gauss-Newton algorithm approximates the function f directly.

1. Init $\theta = \theta_0$.
2. **Iterate**

(a) Solve $\left(\mathbf{J}^T(\theta)\mathbf{J}(\theta)\right)\vec{h} = -\mathbf{J}^T(\theta)\mathbf{f}(\theta)$.

(b) Compute α (using *Line search* or another method).

(c) $\theta = \theta + \alpha\vec{h}$.

(d) Stop, if $\vec{h}$ is small enough.

The algorithm converges, if $\mathbf{J}(\theta)$ has full rank in all iteration steps and if θ_0 is close enough to θ^*. This implies that the choice of the initial parameters not only influences the result (which local minima is found) but also convergence. For θ close to θ^*, convergence becomes quadratic.

The algorithm can be easily be extended to handle sample weights. Let $\mathbf{W}$ be the matrix with the weights in its diagonal $\mathbf{W}_{ii} = w_i$. Then, the linear equation to be solved in the iteration step (1) becomes

$$\left(\mathbf{J}^T(\theta)\mathbf{W}\mathbf{J}(\theta)\right)\vec{h} = -\mathbf{J}^T(\theta)\mathbf{W}\mathbf{f}(\theta)$$

Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm introduces a damping parameter $\mu \geq 0$ to the *Gauss-Newton algorithm*. Using $L(\vec{h}) := \frac{1}{2}\mathbf{l}(\vec{h})^T\mathbf{l}(\vec{h})$, the minimization problem is reformulated as

$$\vec{h} = \text{argmin}_{\vec{h}}\left(L(\vec{h}) + \frac{1}{2}\mu\vec{h}^T\vec{h}\right)$$

and can be computed by solving by the linear equation

$$\left(\mathbf{J}(\theta)^T\mathbf{J}(\theta) + \mu\mathbf{I}\right)\vec{h} = -\mathbf{J}(\theta)^T\mathbf{f}(\theta)$$

From the first equation, it can be seen that the damping parameter adds a penalty proportional to the length of the step $\vec{h}$. Therefore, the step size can be controlled by μ which makes the step size parameter α obsolete. The damping parameter mu can intuitively be described by the following two cases:

1. μ **is large. The linear equation is approximately** $\vec{h} = -\frac{1}{\mu}\nabla(\theta)$**, thus** equivalent to the *Steepest descent* method with small step size.
2. μ **is small (zero in the extreme). The optimization problem becomes more** similar to the original one. This is desired in the final state, when θ is close to θ^*.

Like in *Line search*, the damping parameter can be adjusted iteratively. The gain ratio calculates the ratio between the actual and predicted decrease and is defined as

$$\varrho = \frac{F(\theta) - F(\theta + \vec{h})}{L(\vec{0}) - L(\vec{h}))}$$

where $L(\vec{h})$ is a second order Taylor expansion of F around θ. The denominator can be seen to be $\frac{1}{2}\vec{h}^T\left(\mu\vec{h} - \nabla\right)$. Using this, the damping parameter is set in each step according to the following routine (according to Nielsen):

1. **If** $\varrho > 0$

   (a) $\mu = \mu\max\left(\frac{1}{3}, 1 - (2\varrho - 1)^3\right)$.

(b) $\nu = 2$

2. **Otherwise**

   (a) $\mu = \mu\,\nu$

   (b) $\nu = 2\nu$

with $\nu = 2$, initially. The gain ratio is negative if the step $\vec{h}$ does not actually decrease the target function F. If the gain ratio is below zero as in step (2), the current estimate of θ is not updated.

The written-out **Levenberg-Marquardt algorithm** with adaptive damping parameter looks like this:

1. Initialize $\nu = 2$
2. Initialize $\theta = \theta_0$
3. Initialize $\mu = \tau \max_i \left[\mathbf{J}(\theta_0)^T \mathbf{J}(\theta_0)\right]_{ii}$
4. **Iterate**

   (a) Solve $\left(\mathbf{J}^T(\theta)\mathbf{J}(\theta) + \mu\mathbf{I}\right)\vec{h} = -\mathbf{J}^T(\theta)\mathbf{f}(\theta)$.

   (b) Compute the gain ratio. $\varrho = \frac{F(\theta) - F(\theta+\vec{h})}{L(\vec{0}) - L(\vec{h}))}$

   (c) If the gain ratio is positive, compute μ, ν and update the parameter estimate $\theta = \theta + \vec{h}$

   (d) Otherwise adjust μ, ν

   (e) Stop, if the change in θ or the gradient is small enough.

τ and θ_0 are parameters of the algorithm and therefore have to be provided by the user. The algorithm could also be formulated like the *Gauss-Newton algorithm* with a constant dumping parameter. Yet, this would do not much more than introduce an additional parameter, thus won't be benefitial.

Like in the *Gauss-Newton algorithm*, the optimization problem can also incorporate sample weights:

$$\left(\mathbf{J}^T(\theta)\mathbf{W}\mathbf{J}(\theta) + \mu\mathbf{I}\right)\vec{h} = -\mathbf{J}^T(\theta)\mathbf{W}\mathbf{f}(\theta)$$

Interfaces

class `NLSQ` (*model, derivs, initial=None, lsParams=None, sndDerivs=None*)

Non-linear least-squares base class.

This is an incomplete class for nonlinear least-squares algorithms. The class provides some basic routines that are commonly used.

Subclasses are required to write the *coeff* member after fitting. Otherwise some routine calls may fail or produce wrong results. The default strategy for the step size is a constant.

> **Warning:** Local minima!

Throughout the documentation of this class, the following type symbols are used: - a: The parameters of the model. ???????? - b: The input of the model. May be arbitrary. - c: The output of the model. ????????? - N: Number of training samples. - M: Number of model function parameters.

The search line parameters are: - *rho* ? - *beta* ? - *al_max* ? - *tau* ? - *eps* ?

> **Warning:** parameter dependence!

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *lsParams* ([float]) – Line search parameters
- *sndDerivs* ([[(Num c) :: a -> b -> c]]) – Second order partial derivations of the model function. Expected is a matrix of functions where the element (ij) represents the derivations w.r.t the i'th first and then the j'th parameter. Only required for the *Hessian matrix*.

err (*(x, y)*)
Squared residual.

eval (*x*)
Evaluate the model at data point *x* with coefficients after fitting.

fit (*features, labels=None, weights=None*)
Fit the model to the training data.

Parameters

- *features* – Training samples.
- *labels* – Target values.
- *weights* ([float]) – Sample weights.

Returns self

grad (*theta, data*)
Compute the *Gradient* of the error function.

The error function is dependent on the model function parameters and the training data. This information has to be provided.

Todo

Cannot handle weights. Implemented but commented out (not tested, not quite sure)

Parameters

- *theta* (a) – The parameters of the model function.
- *data* (*STD*) – The training samples.

Returns Gradient as a (Nx1) matrix

hessian (*theta, data*)
Compute the *Hessian matrix* of the error function.

In order to compute the hessian, the first and second order partial derivatives have to be specified (use the constructor's sndDerivs parameter). The error function is dependent on the model function parameters and the training data. This information has to be provided.

Todo

Cannot handle weights. Implemented but commented out (not tested, not quite sure)

Parameters

- *theta* (a) – The parameters of the model function.

- *data* (*STD*) – The training samples.

Returns Hessian matrix (MxM)

jacobian (*theta, data*)
Compute the *Jacobian matrix* of the error function.

The error function is dependent on the model function parameters and the training data. This information has to be provided.

Todo

Cannot handle weights. Actually, as J_ij = [df<j>/dt<i>], weights don't appear In the context of Gauss-Newton and Levenberg-Marquandt, J*W is used.

Parameters

- *theta* (a) – The parameters of the model function
- *data* (*STD*) – The training samples.

Returns Jacobian matrix (NxM)

class SteepestDescent (*model, derivs, initial=None, eps=0.001, maxIter=100, lsParams=None*)
Bases: `ailib.fitting.nlsq.NLSQ`

Implementation of the *Steepest descent* algorithm.

The initial parameter values and convergence threshold have to be set either at object construction or at the call to the *fit* method. The latter overwrites the former.

The default strategy for step size computation is the *Soft line search*. This behaviour can be modified through the *alpha* member. Example:

```
>>> o = SteepestDescent(...)
>>> o.alpha = o._exactLineSearch
```

If the fitting fails due to an OverflowError, try modifying the line search parameters. Consider setting *al_max* to a relatively small value (<< 1.0), although this will increase convergence time.

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.
- *maxIter* (int) – Maximum number of iterations.
- *lsParams* ([float]) – Line search parameters.

fit (*samples, labels, weights=None, theta=None, eps=None*)
Determine the parameters of a model, s.t. it optimally fits the training data.

The parameters *theta* and *eps* are optional. If one of the is not set, its default value determined at object construction will be used (use the constructor's *initial* and *eps* arguments).

Parameters

- *samples* ([]) – Training data.

- *labels* ([]) – Training labels.
- *weights* ([float]) – Sample weights.
- *theta* (a) – Initial parameter choice.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.

Returns self

class **Newton** (*model, derivs, sndDerivs, initial=None, eps=None, maxIter=100, lsParams=None*)
Bases: `ailib.fitting.nlsq.NLSQ`

Implementation of *Newton's method*.

Parameter handling and step size strategy behave as in `SteepestDescent`

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *sndDerivs* ([[(Num c) :: a -> b -> c]]) – Second order partial derivations of the model function.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.
- *maxIter* (int) – Maximum number of iterations.
- *lsParams* ([float]) – Line search parameters

fit (*samples, labels, weights=None, theta=None, eps=None*)
Determine the parameters of a model, s.t. it optimally fits the training data.

The parameters *theta* and *eps* are optional. If one of the is not set, its default value determined at object construction will be used (use the constructor's *initial* and *eps* arguments).

Parameters

- *samples* ([]) – Training data.
- *labels* ([]) – Training labels.
- *weights* ([float]) – Sample weights.
- *theta* (a) – Initial parameter choice.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.

Returns self

class **HybridSDN** (*model, derivs, sndDerivs, initial=None, eps=None, maxIter=100, lsDescent=None, lsNewton=None*)
Bases: `ailib.fitting.nlsq.Newton`

Implementation of the *Hybrid SD/Newton* algorithm.

The *Newton* step is chosen, if the *Hessian matrix* is positive definite. Otherwise, the *steepest descent method* is applied.

Parameter handling and step size strategy behave as in `SteepestDescent`

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *sndDerivs* ([[(Num c) :: a -> b -> c]]) – Second order partial derivations of the model function.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.
- *maxIter* (int) – Maximum number of iterations.
- *lsDescent* ([float]) – Line search parameters for the steepest descent part.
- *lsNewton* ([float]) – Line search parameters for the newton part.

fit (*samples, labels, weights=None, theta=None, eps=None*)
Determine the parameters of a model, s.t. it optimally fits the training data.

The parameters *theta* and *eps* are optional. If one of the is not set, its default value determined at object construction will be used (use the constructor's *initial* and *eps* arguments).

Parameters

- *samples* ([]) – Training data.
- *labels* ([]) – Training labels.
- *weights* ([float]) – Sample weights.
- *theta* (a) – Initial parameter choice.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.

Returns self

class GaussNewton (*model, derivs, initial=None, eps=None, maxIter=100, lsParams=None*)
Bases: `ailib.fitting.nlsq.NLSQ`

Implementation of the *Gauss-Newton algorithm*.

Parameter handling and step size strategy behave as in `SteepestDescent`

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.
- *maxIter* (int) – Maximum number of iterations.
- *lsParams* ([float]) – Line search parameters

fit (*samples, labels, weights=None, theta=None, eps=None*)
Determine the parameters of a model, s.t. it optimally fits the training data.

The parameters *theta* and *eps* are optional. If one of the is not set, its default value determined at object construction will be used (use the constructor's *initial* and *eps* arguments).

Parameters

- *samples* ([]) – Training data.
- *labels* ([]) – Training labels.
- *weights* ([float]) – Sample weights.
- *theta* (a) – Initial parameter choice.
- *eps* (float) – Convergence threshold. The iteration stops, if the step length is below this threshold.

Returns self

class Marquardt (*model, derivs, initial=None, eps0=0.01, eps1=0.01, tau=1.0, maxIter=100*)
Bases: `ailib.fitting.nlsq.NLSQ`

Implementation of the *Levenberg-Marquardt algorithm*.

The damping parameter mu is initialized with *tau* and adjusted according to *Nielsen's strategy*.

Parameters

- *model* ((Num c) :: a -> b -> c) – Model function to be fitted.
- *derivs* ([(Num c) :: a -> b -> c]) – First order derivations of the model function. Expected is list where the i'th element represents the derivation w.r.t the i'th parameter.
- *initial* (c) – Initial choice of parameters. The initial choice may influence result and convergence speed.
- *eps0* (float) – Convergence threshold. The iteration stops, if the maximal gradient component is below this threshold.
- *eps1* (float) – Convergence threshold. The iteration stops, if the relative change in the model parameters is below this threshold.
- *tau* (float) – Constant for determining the initial damping. Use a small value (10**(-6)), if the initial parameters are believed to be a good approximation, otherwise use 10**(-3) or 1.0 (default)
- *maxIter* (int) – Maximum number of iterations.

fit (*samples, labels, weights=None, theta=None, eps0=None, eps1=None, tau=None*)
Determine the parameters of a model, s.t. it optimally fits the training data.

The parameters *theta*, *eps0*, *eps1* and *tau* are optional. If one of the is not set, its default value determined at object construction will be used (use the constructor's *initial*, *eps0*, *eps1* and *tau* arguments).

Parameters

- *samples* ([]) – Training data.
- *labels* ([]) – Training labels.
- *weights* ([float]) – Sample weights.
- *theta* (a) – Initial parameter choice.

- *eps0* (float) – Convergence threshold. The iteration stops, if the maximal gradient component is below this threshold.
- *eps1* (float) – Convergence threshold. The iteration stops, if the relative change in the model parameters is below this threshold.
- *tau* (float) – Constant for determining the initial damping. Use a small value (10**-6), if the initial parameters are believed to be a good approximation, otherwise use 10^-3 or 1.0 (default)

Returns self

Examples

k-Nearest Neighbour

Contents

The idea of k-Nearest neighbour classification very simple. First some training data has to be provided. Then, any new data point is assigned the same label as the majority of its k closest training data points. In the simplest case, this looks like this:

1. Store all training points
2. When a new data point is presented, compute the distance to all training points. As default, the euclidean distance is used, i.e.

$$d_i(\nu) = \|\nu - x_i\|^2$$

3. Select the training samples with k smallest distances.
4. Return the label of the selected training samples that appeared most often.

If sample weights are given, the distance in the second step is modified:

$$d_i(\nu) = \frac{1}{N w_i}\|\nu - x_i\|^2$$

This adjustment makes it more probable for a sample with a large weight to be in the set of nearest neighbours. Instead of a majority vote on the labels, it is also possible to pick a label at random (with respect to the distance to that point).

Only the base algorithm is implemented here. If a more advanced implementation is required, have a loot at the scikit-learn package.

Interfaces

class `kNN`(*k, dist=<function <lambda> at 0x36ec140>*)

Bases: `ailib.fitting.model.Model`

k-Nearest Neighbour matching.

A data point *x* is assigned to the same class as the most of its k nearest neighbours.

if weights are given, the distance will be computed as

```
>>> 1.0/(N*weight[i]) * dist(data[i],x)
```

There are three evaluation types:

- The most often appeared label (majority vote, the default).
- Sampled from all neighbours with uniform probability (mixin Sampling).
- Sampled from all neighbours with respect to their distances (mixin WeightedSampling).

The distance measurements

- *distAbs*: absolute distance $|x - y|$.
- *distNorm*: norm $\|x - y\|$.
- *distSqNorm*: squared norm $\|x - y\|^2$.

are made available. Change the distance function by assignment to *obj.dist* For the lower ones, the data should be passed as list or a scipy.matrix. The default is distSqNorm.

Parameters

- *k* (int) – Parameter which specifies the size of the neighbourhood.
- *dist* ((Num b) => a -> a -> b) – Distance measurement on data points.

class **Sampling**()
Mixin for sampling with uniform probabilities.

eval(*x*)
Samples the label from all k nearest neighbours.

All neighbours are equally likely to be drawn.

class **WeightedSampling**()
Mixin for sampling with respect to distances.

eval(*x*)
Samples the label from all k nearest neighbours with respect to their distance.

The sampling weights are inverse proportional to the distance:

```
>>> w[i] = 1.0/(dist(data[i],x))
```

with weights enabled

```
>>> w[i] = N * weights[i] / dist(data[i], x)
```

err(*(x, y)*)
0-1 loss function.

eval(*x*)
Return the label of the most frequent class of the k nearest neighbours of *x*.

If there are two classes with the same number of occurrences in the k-neighbourhood of *x*, one of them is chosen at random (i.e. ties are broken randomly).

fit (*data, labels, weights=None*)
Set up the data points for later matching.

Examples

Bagging

Contents

Introduction

Bagging is mainly used to reduce the prediction variance. Given several classifiers, the bagging approach can be formulated as follows:

1. For all classifiers, do
   (a) Draw a bootstrap sample set, given all training samples. Bootstrapping means to draw N samples with replacement from the N original training samples. This implies that all classifiers see the same number of samples but samples may be identical and different classifiers are presented different sample sets.
   (b) Train the classifier on the bootstrapped sample set.
2. Evaluate a new data point by evaluating all models and then form some kind of consensus answer (majority vote for classification and average for regression problems).

Bagging is independent on the type of the trainable models (i.e. can handle classification and regression). See the `implementation` for details.

Interfaces

class Bagging()
Bases: `ailib.fitting.model.Committee`

The Bagging algorithm is implemented as extension of a committee of models. Specifically, the committee is extended by a simple fitting method.

As the evaluation (and error computation) is dependent on the specific problem, use the `Committee` mixins like so:

```
>>> class foo(Bagging, Committee.Classification): pass
```

The models have to be assigned to the instance before training. For this, the method `Committee.addModel()` can be used.

fit (*data*)
Fit the assigned models to sets, *bootstrapped* from *data*.

Examples

Boosting

AdaBoost

Given N training data points and M classifiers. The AdaBoost algorithm does the following:

1. Initialize weights $w_i = \frac{1}{N}$
2. For $m = 1..M$:
    (a) Fit the classifier G_m to the training data using weights w_i
    (b) Compute the classifier error

$$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I\{y_i \neq G_m(x_i)\}}{\sum_{i=1}^{N} w_i}$$

    (a) Compute the classifier weight

$$\alpha_m = \log\left(\frac{1 - \text{err}_m}{\text{err}_m}\right)$$

    (a) Recompute all data weights w_i

$$w_i = w_i \exp\left(\alpha_m I\{y_i \neq G_m(x_i)\}\right)$$

3. The combined classifier is

$$G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

Interfaces

class `AdaBoost`()

Bases: `ailib.fitting.model.Model`

AdaBoost algorithm.

Given M (weak) classifiers, AdaBoost iteratively trains each one of them with respect to so far misclassified samples. For a formal representation, consult the respective *documentation*.

As can be seen from the schematics, the classifiers are required to support weighted samples. The classifiers have to be linked to the instance of this class before training. This can be achieved through calls to the `AdaBoost.addModel()` method.

addModel (*m*)
Add a trainable model to the set of boosted classifiers.

The models are required to support weighted samples.

Parameter *m* (Model) – Model

err (*(x, y)*)
Return the distance between the target *y* and the model prediction at the point *x*.

eval (*x*)
Evaluate the model at data point *x*.

The outcome is the majority vote over the outcome of all classifiers w.r.t. their weight (note the *formal representation*).

fit (*data*)
Apply the AdaBoost algorithm on the presented data set.

Parameter *data* – Training data.

Returns self

Examples

Trees and Random Forests

Contents

Decision tree

The tree growing algorithm can be sketched as follows:

1. Grow the tree
   (a) Given the data, determine the optimal dimension *j* and splitting point *s*.

$$(j^*, s^*) = \min_{j,s} \left(\min_{c_1} \sum_{i:\, x_i[j] \le s} (y_i - c_1)^2 + \min_{c_2} \sum_{i:\, x_i[j] > s} (y_i - c_2)^2 \right)$$

   where c_1 and c_2 are the labels of the (hypothetical) leaves.
   (a) Grow two child trees on the respective data subsets.
   (b) Stop, if the number of data points is below some threshold.

2. Prune the tree
   (a) Collapse the internal node that produces the smallest per-node increase in the tree cost.
   (b) Stop, if there's only one node left (the root).
   (c) Out of the produced sequence of trees, select the one that minimizes the total tree cost.

Stumps are single-node trees with the node labels fixed to $\pm m$. Also for stumps, the learning consists of finding the optimal splitting dimension j and threshold s. Because stumps are classifiers, the *0-1 loss function* is used as error measurement. Thus, each stump has to solve the optimization problem:

$$(j^*, s^*) := \min_{j,s} \frac{\sum_{i=1}^{N} w_i I\{y_i \neq c(x_i|j,s,m)\}}{\sum_{i=1}^{n} w_i}$$

Additionally to the training input, each training sample (x_i, y_i) is assigned a weight w_i that represents how much a misclassification of the sample contributes to the classifier cost.

Random forest

Interfaces

class `Tree`(*leafThres=5*)
Bases: `ailib.fitting.model.Model`

Decision tree skeleton.

> **Warning:** Don't use this class directly. Use `RegressionTree` or `ClassificationTree` instead.

Parameter *leafThres* (int) – Minimum number of data points per node.

`cleanData`()
Remove the data from the tree.

Only the data member is cleared. The node label is recomputed first.

`collude`()
Collapses the childs of the node.

The childs' data is collected in this node and the label recomputed. The childs will be deleted. The node will become a leaf.

`costSplit`(*j, s*)

`costTerminal`(*data*)

`costTree`(*alpha*)

`data`()
Return the data of all child nodes.

`depth`()
Return the maximal depth of the tree.

`eval`(*x*)
Find the label of a data point *x*.

grow (*data*)
Grow a tree on training data.

Finds the optimal splitting parameters of the data and grows two child trees (if there are enough data points left).

Parameter *data* (*STD*) – Training data.

isLeaf ()
Return true, if the node has no childs (is a leaf node).

leaves ()
Return the number of leaves.

prune (*alpha*)
Prune the tree to find the tree with minimal cost.

The tree is pruned according to weakest link pruning until there's only one node left. Then, the tree with minimal total cost is restored.

Parameter *alpha* (float) – Pruning parameter. Controls, how much the tree size influences its cost.

class Stump (*labelValue=1.0*)
Bases: `ailib.fitting.model.Model`

Parameter *labelValue* (float) – Label of the class.

err (*(x, y)*)
0-1 loss function.

eval (*x*)
Compute the label of a data point *x*.

fit (*data, weights=None*)
Train the stump on the presented data.

Finds the optimal dimension and splitting point to split the data in two subsets. The optimal parameters are found w.r.t. to the data weights. If no weights are given, a uniform distribution is assumed.

Parameters

- *data* (*STD*) – Data points.
- *weights* ([float]) – Weights of data points. Needs not to be normalized.

Classification

class ClassificationTree (*alpha=0.0, leafThres=5*)
Bases: `ailib.fitting.tree.Tree`

Parameters

- *alpha* (float) –
- *leafThres* (int) –

costSplit (*data, j, s*)

costTerminal (*data*)

costTree (*alpha*)

err (*(x, y)*)

fit (*data*)

Regression

class `RegressionTree`(*alpha=0.0, leafThres=5*)

Bases: `ailib.fitting.tree.Tree`

Parameters

- *alpha* (float) –
- *leafThres* (int) –

`costSplit`(*data, j, s*)

`costTerminal`(*data*)

`costTree`(*alpha*)

`err`(*(x, y)*)

`fit`(*data*)

Examples

Unsupervised learning

Clustering

Contents

K-Means

The K-Means algorithm is used for clustering of unsupervised data.

Classically, the algorithm is sketches as follows:

1. Initialize cluster centers at randomly chosen data points.
2. Iterate until convergence
   (a) Assign each data point to the closest cluster (E-Step).
   (b) Recompute the cluster centers as the average of all associated data points (M-Step).
3. A new data point is classified according to the closest cluster center.

As also hinted by the above description, the K-Means algorithm is an instance of an *Expectation-Maximization* algorithm. The problem can be formalized as

$$\arg\min_{c,y} \frac{1}{N} \sum_{i}^{N} \|x_i - y_{c(i)}\|^2$$

where x are the data points, y the cluster centers and $c(i)$ an assignment function that stores the cluster index for each sample. Setting the derivative w.r.t. the cluster centers to zero yields the recomputation (M) step. This is the stationary condition, so assignments are fixed. In the E-Step, the opposite is the case: the assignments are optimized while the center positions are constant. This yields the optimization problem

$$c(i) = \arg\min_j \|x_i - y_j\|^2$$

Obviously, the solution is to assign a data point to the closest cluster is the E-Step from the algorithm.

In this library, instead of hard assignments like in the above algorithm, the data can be assigned to the clusters in a probabilistic way. The algorithm then becomes:

1. Initialize assignment probability matrix: $\mathbf{P}_{i,\alpha} = \frac{1}{N}$.
2. Iterate until convergence
   (a) Compute the assignment probabilities (E-Step):

$$\mathbf{P}_{i,\alpha} = \frac{\exp\left(-(x_i - y_\alpha)^2/T\right)}{\sum_{\nu=1}^{K} \exp\left(-(x_i - y_\nu)^2/T\right)}$$

   (b) Recompute the cluster centers (M-Step):

$$y_\alpha = \frac{\sum_{i=1}^{N} \mathbf{P}_{i,\alpha} x_i}{\sum_{i=1}^{N} \mathbf{P}_{i,\alpha}}$$

3. For a new data point x, compute the assignment probabilities

$$\mathbf{P}(\alpha) = \frac{\exp\left(-(x - y_\alpha)^2/T\right)}{\sum_{\nu=1}^{K} \exp\left(-(x - y_\nu)^2/T\right)}$$

   and sample from the discrete distribution formed by $\{\mathbf{P}(\alpha) \,|\, \alpha \in [1, K]\}$.

The matrix $\mathbf{P}$ stores the assignment probabilities to all clusters for each sample. Note that $\sum_\alpha \mathbf{P}_{i,\alpha} = 1$. This representation comes from a maximum entropy approach.

The risk is defined as in the 'normal' algorithm. The E-Step assumes fixed cluster centers, so the risk can be expressed as:

$$R(\alpha) = \sum_i (x_i - y_\alpha)^2$$

and inserted into the gibbs distribution (which maximizes the entropy) with parameter T.

$$\mathbf{P}(c; y_\nu) = \frac{\exp(-R(c)/T)}{\sum_{c' \in \mathcal{C}} \exp(-R(c')/T)}$$

Note that $\mathcal{C}$ denotes the set of all possible assignments. The equation can be reformulated to give the probability of an assignment c (given the cluster centers y_ν) and from there, the assignment probabilities $\mathbf{P}_{i,\alpha}$ used in the E-Step are found. Further, the entropy maximization for the cluster center is:

$$y_\alpha = \arg\max_{y_\nu} - \sum_{c \in \mathcal{C}} \mathbf{P}(c; y_\nu) \log \mathbf{P}(c; y_\nu)$$

This equation can be again solved using the stationary condition (setting the second derivative w.r.t. y_ν to zero) and gets the M-Step of the algorithm.

In this representation, the parameter T is introduced. A large temperature T leads to fewer diverse centers and uniform assignment probability of the data to the centers. A small temperature results in the inverse effect: The centers become distinct and the assignment probability of each sample tends towards one for one cluster and zero for the others. Loosely speaking, the temperature controls how strongly the clusters are influenced by the data, i.e how sensitive the algorithm is to the input data.

Interfaces

class **Kmeans** (*numClusters, tol=1, maxIter=100, temp=1.0, dist=<function <lambda> at 0x36eacf8>*)

Bases: `ailib.fitting.kNN.kNN`

K-Means algorithm.

This is an implementation of the probabilistic K-Means algorithm, not the often used Lloyd algorithm. The default evaluation method is to assign the sample to the closest cluster. Instead, the class label can be sampled from the distances, i.e. the closer to a cluster, the more probable this cluster label will be returned. If this behaviour is desired, use the `Kmeans.Soft` mixin.

There are several convergence criterions:

- Risk is below a threshold (*absoluteRisk*).
- risk decrease is below a threshold (*relativeRisk*).
- Stop after a number of iterations (*iterCriterion*).
- Assignments don't change (*centerCriterion*).

Select one of those by assignment to *obj._converged*.

Parameters

- *numClusters* (int) – Number of clusters.
- *tol* (float) – Threshold for the convergence criterion.
- *maxIter* (int) – Limit to number of iterations.
- *temp* (float) – Temperature.
- *dist* (a -> a -> float) – Distance measurement

class **Soft** ()

Sample from clusters according to their distances.

eval (*x*)

Return the class of *x*, sampled from the available clusters.

fit (*data, labels=None, weights=None, T=None*)

Parameters

- *data* ([a]) – Training data.
- *labels* (None) – Unused.
- *weigths* – Unused.
- *T* – Temperature. If set, overwrites member.

Returns self

Examples

1.5 `selection` — Model selection and validation

Contents

1.5.1 Information Criteria

In *maximum likelihood* methods, the Bayesian and Akaike information criteria give a measurement

Let $L = \max_{\theta} p(\theta|\vec{x}, \vec{y})$ be the likelihood of the most probable model parameters θ with respect to the training data $\vec{x}, \vec{y}$. Furthermore, let the model have k free parameters that are estimated and let there be n training samples. The two information criteria are defined in the following way:

1. Bayesian information criterion

$$\text{BIC} := -2\ln(L) + k\ln(n)$$

2. Akaike information criterion

$$\text{AIC} := -2\ln(L) + 2k$$

From this definition, it can be seen that the criteria add a complexity penalty to the parameter likelihood. The reason behind this is the following: it's assumed that the higher the number of parameters, the better any set of training data can be approximated by the model. For example, consider polynomial fitting. The number of free parameters corresponds to the order of the fitted polynomial. Given a degree equal to the number of training samples, the polynomial can be fitted perfectly. But as it will be a curve through all given data pairs, the generalization power will be poor. The goal is to find a model with reasonable number of parameters but at the same time a low training error (i.e. a high maximum likelihood).

If the error is normal *i.i.d* with variance $\hat{\sigma}^2$, the criteria can be stated as:

$$\begin{aligned} \text{BIC} &= n \ln \hat{\sigma}^2 + k \ln(n) \\ \text{AIC} &= n \ln \hat{\sigma}^2 + 2k \end{aligned}$$

Instead of the Akaike information criterion, one should rather empoly the corrected AIC:

$$\text{AICc} := \text{AIC} + \frac{2k(k+1)}{n-k-1}$$

As n gets large, the two criteria are identical. If n is relatively small, the original AIC may suggest models with a larger number of parameters, thus is more likely to be subject of *overfitting*.

1.5.2 K-Fold Cross-Validation

The cross-validation method tries to estimate the *generalization error* of a model, given some training data. For K-fold cross-validation, the N samples are split into K subsets of equal size. Then, $K-1$ subsets are used as *training set* and the remaining subset as *testing set*:

1. Split the data into K subsets.
2. For $k = 1..K$
   (a) Fit the model using all subsets except the k'th.
   (b) Compute the testing error err_k on the k'th subset.
3. The estimated generalization error is the average of the errors of the K testing sets.

$$err_{CV} = \frac{1}{K} \sum_{k=1}^{K} err_k$$

If $K = N$, each sample is once used as *testing set* (consisting of only this sample). This special case is called **Leave-one-out cross-validation**.

1.5.3 Random Sample Consensus

Given a set of training data that is noisy and contains outliers, i.e. erroneous data. The aim of the RANSAC algorithm is to determine which data points are outliers and compute the model without those.

Given the minimal number of points M required to fit the model and a threshold, the algorithm does the following:

1. Iterate until convergence
   (a) Out of all data points, select M points randomly (the root set).
   (b) Fit the model to these.
   (c) Compute the prediction error for all data points.
   (d) Add all points for which the error is below a threshold to the consensus set of this iteration.
2. Fit the model using all points in the largest consensus set.

As convergence criterion, the probability that at least one outlier free root set was chosen can be tracked. Let s be the number of iterations and r the ratio of inliers and outliers.

$$p = 1 - (1 - r^M)^s$$

Because r is not known beforehand, it is estimated by $\frac{\text{\# inliers in the best model}}{\text{\# data points}}$. The algorithm iterates, until p exceeds some value (usually close to one, e.g. $p > 0.99$).

1.5.4 Interfaces

crossValidation(*data, model, K*)

Parameters

- *data* (*STD*) – Data points.
- *model* (ailib.fitting.Model) – Model to fit the data points to.
- *K* (int) – Number of buckets.

Returns Cross-validation error.

class Ransac(*thres, minPts, prob=0.98999999999999999, maxIter=100*)

Parameters

- *thres* (float) – Error threshold.
- *prob* (float) – Minimum probability of finding an outlier-free root set.
- *maxIter* (int) – Maximum number of iterations.
- *minPts* (int) – Minimum number of points the model requires to be computed.

fit(*data, model*)

Parameters

- *data* (*STD*) – Training data.
- *model* (ailib.fitting.Model) – Model to be fit.

Returns *model* instance, fitted to the optimal consensus set.

1.5.5 Examples

1.6 Glossary

0-1 loss function The cost of a missclassification is one, the cost of a correct classification zero.:

```
if label == prediction:
    cost = 0.0
else:
    cost = 1.0
```

In mathematical formulations, this can also be written as:

$$I\{y_i \neq c(x_i)\}$$

Bootstrapped see *Bootstrapping*

Bootstrapping Given N samples, the bootstrapped dataset contains N points randomly drawn with replacement from the original dataset.

Classification In classification problems, an input feature has to be assigned to a certain class. Often, classes represent some attributes or characteristics of the possible input, i.e. divide the input space in discrete regions. Classification can also be seen as a coding scheme, where an input string has to be assigned to one codebook entry.

Data point Training data point, usually represented as a *feature*.

Expectation-Maximization This term refers to an iterative algorithm scheme, consisting of two steps. It is applied in maximum likelihood methods, where some information is missing (hidden). In the E-step, the hidden states are estimated according to the currently computed model. Then, the model is again recomputed using the estimated hidden states in the M-step. These two steps are repeated, until the procedure converged.

Feature Description of a data point. Represented by a vector $[x_0, x_1, \ldots, x_n]$

Generalization error The generalization error expresses how well a trained model behaves on any possible (so far not seen) input. Loosely speaking, it shows how well the model represents the true underlying problem. In general, it cannot be measured nor computed, at most estimated.

Gradient Given a function $F(\vec{x})$ with $\vec{x}$ being a vector of variables x_i. Then, the gradient is the vector of all first order partial derivatives w.r.t each component of $\vec{x}$:

$$\nabla(\vec{x}) := \left[\frac{\partial F}{\partial x_0}(\vec{x}), \frac{\partial F}{\partial x_1}(\vec{x}), \ldots, \frac{\partial F}{\partial x_n}(\vec{x})\right]^T$$

Hessian matrix

$$H(\vec{x}) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j}(\vec{x})\right]_{ij}$$

i.i.d Identically, independent distributed

Jacobian matrix Given a function with parameters x_i and input values a_j. The Jacobian matrix holds the first order partial derivative w.r.t every parameter, evaluated at each input value:

$$\mathbf{J}(a)_{ij} = \left[\frac{\partial f}{\partial x_i}(a_j)\right] = \begin{pmatrix} \frac{\partial f(a_0)}{\partial x_0} & \frac{\partial f(a_0)}{\partial x_1} & \cdots & \frac{\partial f(a_0)}{\partial x_M} \\ \frac{\partial f(a_1)}{\partial x_0} & \frac{\partial f(a_1)}{\partial x_1} & \cdots & \frac{\partial f(a_1)}{\partial x_M} \\ \cdots & & & \\ \frac{\partial f(a_N)}{\partial x_0} & \frac{\partial f(a_N)}{\partial x_1} & \cdots & \frac{\partial f(a_N)}{\partial x_M} \end{pmatrix}$$

Label In supervised learning situations, the label specifies the optimal model outcome. In classification problems, the label is a class identifier, in regression problems it's the target value (in the function's range).

Maximum likelihood Given a model with some - so far unspecified - parameters, the maximum likelihood estimation denotes a method to determine the parameters optimally, given training data.

Model The core of each learning problem is the model. Given some training data and a learning goal, the model is a mathematical representation that captures the shape or structure of the underlying problem. The model can usually be trained, i.e. fitted to the training data, and produce an output that can be mapped to the learning goal.

Overfitting In many learning problems, one desires to come up with a model that minimizes the prediction error on a training set. Unfortunately, models with higher complexity often decrease the training error but rely too much on the concrete data set.

As an example, consider polynomial fitting. A high order polynomial fits any training set well but also includes all noise from the data in the model. If the data (say 100 data points) is noisily distributed along a parabola, a polynom of high order (e.g. 100) will give a low training error but isn't capable of capturing the actually relevant information (the shape of the parabola).

The term overfitting describes a situation where the complexity of the chosen model does not match the complexity of the actual underlying problem, resulting in a trained model that only works for the training data. In general, such situations cannot be detected automatically (as the training error is actually minimal). Typically the testing error will be very low while the generalization error is high.

Regression Regression problems deal with adapting a continous function to measured training data. Often, the target function given in a general form, including parameters. The goal of regression is to determine parameter values, such that the training data is optimally represented by the function.

STD Supervised training data: [(*feature*, label)]

Testing error The training error identifies the error statistics of a trained model over all data points in the testing set. Usually, the testing set is a subset of the originally measured data, not used for training. As the testing set is limited, the testing error cannot be equal to the generalization error. Yet, it is used to give a hint how well the trained model will behave on new data.

Testing set In order to measure the *testing error*, a set of data is needed that was not already seen by the model, i.e. not used to train (fit) the model. Hence, the part of available data which is reserved for this task is called the testing set. Testing data is only available in the case of supervised learning (yet, there may be exceptions). The format is determined by *STD*.

Training error The training error identifies the error statistics of a trained model over all data points in the training set.

Training set The testing set is the collection of data which is used to train (fit) a model. For supervised learning, the training set is a list of tuples (the *feature* and *label*). In the case of unsupervised learning, the training set is simply the list of *feature*s. (See also *STD* and *UTD*).

UTD Unsupervised training data: [*feature*]

1.7 Download

Contents

1.7.1 Installation

Using Pip Installs Python (Pip), simply type:

```
pip install http://www.igsor.net/aiLib/_downloads/latest.tar.gz
```

if you want to use the package from the webpage. If you have downloaded it yourself, use:

```
pip install path/to/aiLib.tar.gz
```

If you're using distutils, type:

```
tar -xzf path/to/aiLib.tgz      # extract files.
cd aiLib*                       # change into aiLib directory.
sudo python setup.py install    # install using distutils (as root).
#rm -R aiLib*                   # remove source. If desired, uncomment this line.
```

Make sure, scipy and numpy are installed on your system.

1.7.2 Getting started

Again, make sure that besides the aiLib, the packages scipy and numpy are installed on your system.

1.7.3 Available downloads

- **aiLib-0.1 dev** (latest)
- **This documentation (html)** (current)
- **This documentation (pdf)** (current)

1.7.4 License

This project is released under the terms of the 3-clause BSD License. See the section *License* for details.

1.8 License

This project is released under the terms of the 3-clause BSD License.

```
Copyright (c) 2012, Matthias Baumgartner
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the aiLib nor the
      names of its contributors may be used to endorse or promote products
      derived from this software without specific prior written permission.
```

1.9 Resources

CHAPTER

TWO

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[ESLII] T. Hastie, R. Tibshirani, J. Friedman *Elements of statistical learning*, 2nd edition Web

[IMM3215] P.E. Frandsen, K. Jonasson, H.B. Nielsen, O. Tingleff *Unconstrained optimization*, 3rd Edition 2004 **download**

[IMM3217] K. Madsen, H.B. Nielsen, O. Tingleff *Methods for non-linear least squares problem*, 2nd Edition 2004 **download**

MODULE INDEX

A

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V