MultivariateStats Documentation

Release 0.1.0

Dahua Lin

Apr 23, 2017

Contents

MultivariateStats.jl is a Julia package for multivariate statistical analysis. It provides a rich set of useful analysis techniques, such as PCA, CCA, LDA, PLS, etc.

Contents:

CHAPTER 1

Linear Least Square and Ridge Regression

The package provides functions to perform *Linear Least Square* and *Ridge Regression*.

Linear Least Square

Linear Least Square is to find linear combination(s) of given variables to fit the responses by minimizing the squared error between them. This can be formulated as an optimization as follows:

$$\underset{(\mathbf{a},b)}{\text{minimize}} \ \frac{1}{2}\|\mathbf{y} - (\mathbf{X}\mathbf{a} + b)\|^2$$

Sometimes, the coefficient matrix is given in a transposed form, in which case, the optimization is modified as:

$$\underset{(\mathbf{a},b)}{\text{minimize}} \ \frac{1}{2}\|\mathbf{y} - (\mathbf{X}^T\mathbf{a} + b)\|^2$$

The package provides `llsq` to solve these problems:

`llsq`(*X*, *y*; ...)

Solve the linear least square problem formulated above.

Here, `y` can be either a vector, or a matrix where each column is a response vector.

This function accepts two keyword arguments:

- `trans`: whether to use the transposed form. (default is `false`)
- `bias`: whether to include the bias term `b`. (default is `true`)

The function results the solution `a`. In particular, when `y` is a vector (matrix), `a` is also a vector (matrix). If `bias` is true, then the returned array is augmented as `[a; b]`.

Examples

For a single response vector `y` (without using bias):

```
using MultivariateStats

# prepare data
X = rand(1000, 3)                # feature matrix
a0 = rand(3)                     # ground truths
y = X * a0 + 0.1 * randn(1000)   # generate response

# solve using llsq
a = llsq(X, y; bias=false)

# do prediction
yp = X * a

# measure the error
rmse = sqrt(mean(abs2(y - yp)))
print("rmse = $rmse")
```

For a single response vector `y` (using bias):

```
# prepare data
X = rand(1000, 3)
a0, b0 = rand(3), rand()
y = X * a0 + b0 + 0.1 * randn(1000)

# solve using llsq
sol = llsq(X, y)

# extract results
a, b = sol[1:end-1], sol[end]

# do prediction
yp = X * a + b
```

For a matrix `Y` comprised of multiple columns:

```
# prepare data
X = rand(1000, 3)
A0, b0 = rand(3, 5), rand(1, 5)
Y = (X * A0 .+ b0) + 0.1 * randn(1000, 5)

# solve using llsq
sol = llsq(X, Y)

# extract results
A, b = sol[1:end-1,:], sol[end,:]

# do prediction
Yp = X * A .+ b
```

Ridge Regression

Compared to linear least square, Ridge Regression uses an additional quadratic term to regularize the problem:

$$\underset{(\mathbf{a},b)}{\text{minimize}} \ \frac{1}{2}\|\mathbf{y} - (\mathbf{X}\mathbf{a} + b)\|^2 + \frac{1}{2}\mathbf{a}^T\mathbf{Q}\mathbf{a}$$

The transposed form:

$$\underset{(\mathbf{a},b)}{\text{minimize}} \ \frac{1}{2}\|\mathbf{y} - (\mathbf{X}^T\mathbf{a} + b)\|^2 + \frac{1}{2}\mathbf{a}^T\mathbf{Q}\mathbf{a}$$

The package provides `ridge` to solve these problems:

`ridge` (*X*, *y*, *r; ...*)

Solve the ridge regression problem formulated above.

Here, `y` can be either a vector, or a matrix where each column is a response vector.

The argument `r` gives the quadratic regularization matrix `Q`, which can be in either of the following forms:

- `r` is a real scalar, then `Q` is considered to be `r * eye(n)`, where `n` is the dimension of `a`.
- `r` is a real vector, then `Q` is considered to be `diagm(r)`.
- `r` is a real symmetric matrix, then `Q` is simply considered to be `r`.

This function accepts two keyword arguments:

- `trans`: whether to use the transposed form. (default is `false`)
- `bias`: whether to include the bias term `b`. (default is `true`)

The function results the solution `a`. In particular, when `y` is a vector (matrix), `a` is also a vector (matrix). If `bias` is true, then the returned array is augmented as `[a; b]`.

CHAPTER 2

Data Whitening

A whitening transformation is a decorrelation transformation that transforms a set of random variables into a set of new random variables with identity covariance (uncorrelated with unit variances).

In particular, suppose a random vector has covariance $\mathbf{C}$, then a whitening transform $\mathbf{W}$ is one that satisfy:

$$\mathbf{W}^T\mathbf{C}\mathbf{W} = \mathbf{I}$$

Note that $\mathbf{W}$ is generally not unique. In particular, if $\mathbf{W}$ is a whitening transform, so is any of its rotation $\mathbf{W}\mathbf{R}$ with $\mathbf{R}^T\mathbf{R} = \mathbf{I}$.

Whitening

The package uses `Whitening` defined below to represent a whitening transform:

```
immutable Whitening{T<:FloatingPoint}
    mean::Vector{T}      # mean vector (can be empty to indicate zero mean), of length␣
↪d
    W::Matrix{T}         # the transform coefficient matrix, of size (d, d)
end
```

An instance of `Whitening` can be constructed by `Whitening(mean, W)`.

There are several functions to access the properties of a whitening transform `f`:

indim (*f*)
: Get the input dimension, *i.e* `d`.

outdim (*f*)
: Get the out dimension, *i.e* `d`.

mean (*f*)
: Get the mean vector.

  Note: if `f.mean` is empty, this function returns a zero vector of length `d`.

transform(*f*, *x*)
Apply the whitening transform to a vector or a matrix with samples in columns, as $\mathbf{W}^T(\mathbf{x} - \boldsymbol{\mu})$.

Data Analysis

Given a dataset, one can use the `fit` method to estimate a whitening transform.

fit(*Whitening*, *X; ...*)
Estimate a whitening transform from the data given in `X`. Here, `X` should be a matrix, whose columns give the samples.

This function returns an instance of `Whitening`.

Keyword Arguments:

name	description	default
regcoef	The regularization coefficient. The covariance will be regularized as follows when `regcoef` is positive: `C + (eigmax(C) * regcoef) * eye(d)`	`zero(T)`
mean	The mean vector, which can be either of: •`0`: the input data has already been centralized •`nothing`: this function will compute the mean •a pre-computed mean vector	`nothing`

Note: This function internally relies on `cov_whiten` to derive the transformation `W`. The function `cov_whiten` itself is also a useful function.

cov_whitening(*C*)
Derive the whitening transform coefficient matrix `W` given the covariance matrix `C`. Here, `C` can be either a square matrix, or an instance of `Cholesky`.

Internally, this function solves the whitening transform using Cholesky factorization. The rationale is as follows: let $\mathbf{C} = \mathbf{U}^T\mathbf{U}$ and $\mathbf{W} = \mathbf{U}^{-1}$, then $\mathbf{W}^T\mathbf{C}\mathbf{W} = \mathbf{I}$.

Note: The return matrix `W` is an upper triangular matrix.

cov_whitening(*C*, *regcoef*)
Derive a whitening transform based on a regularized covariance, as `C + (eigmax(C) * regcoef) * eye(d)`.

In addition, the package also provides `cov_whiten!`, in which the input matrix `C` will be overwritten during computation. This can be more efficient when `C` is no longer used.

invsqrtm(*C*)
Compute `inv(sqrtm(C))` through symmetric eigenvalue decomposition.

CHAPTER 3

Principal Component Analysis

Principal Component Analysis (PCA) derives an orthogonal projection to convert a given set of observations to linearly uncorrelated variables, called *principal components*.

This package defines a `PCA` type to represent a PCA model, and provides a set of methods to access the properties.

Properties

Let `M` be an instance of `PCA`, `d` be the dimension of observations, and `p` be the output dimension (*i.e* the dimension of the principal subspace)

`indim`(*M*)
: Get the input dimension `d`, *i.e* the dimension of the observation space.

`outdim`(*M*)
: Get the output dimension `p`, *i.e* the dimension of the principal subspace.

`mean`(*M*)
: Get the mean vector (of length `d`).

`projection`(*M*)
: Get the projection matrix (of size `(d, p)`). Each column of the projection matrix corresponds to a principal component.

  The principal components are arranged in descending order of the corresponding variances.

`principalvars`(*M*)
: The variances of principal components.

`tprincipalvar`(*M*)
: The total variance of principal components, which is equal to `sum(principalvars(M))`.

`tresidualvar`(*M*)
: The total residual variance.

tvar(*M*)
The total observation variance, which is equal to `tprincipalvar(M) + tresidualvar(M)`.

principalratio(*M*)
The ratio of variance preserved in the principal subspace, which is equal to `tprincipalvar(M) / tvar(M)`.

Transformation and Construction

Given a PCA model `M`, one can use it to transform observations into principal components, as

$$\mathbf{y} = \mathbf{P}^T(\mathbf{x} - \boldsymbol{\mu})$$

or use it to reconstruct (approximately) the observations from principal components, as

$$\tilde{\mathbf{x}} = \mathbf{P}\mathbf{y} + \boldsymbol{\mu}$$

Here, $\mathbf{P}$ is the projection matrix.

The package provides methods to do so:

transform(*M*, *x*)
Transform observations `x` into principal components.

Here, `x` can be either a vector of length `d` or a matrix where each column is an observation.

reconstruct(*M*, *y*)
Approximately reconstruct observations from the principal components given in `y`.

Here, `y` can be either a vector of length `p` or a matrix where each column gives the principal components for an observation.

Data Analysis

One can use the `fit` method to perform PCA over a given dataset.

fit(*PCA*, *X;* ...)
Perform PCA over the data given in a matrix `X`. Each column of `X` is an observation.

This method returns an instance of `PCA`.

Keyword arguments:

Let `(d, n) = size(X)` be respectively the input dimension and the number of observations:

name	description	default
method	The choice of methods: •`:auto`: use `:cov` when `d < n` or `:svd` otherwise •`:cov`: based on covariance matrix •`:svd`: based on SVD of the input data	`:auto`
maxoutdim	Maximum output dimension.	`min(d, n)`
pratio	The ratio of variances preserved in the principal subspace.	`0.99`
mean	The mean vector, which can be either of: •`0`: the input data has already been centralized •`nothing`: this function will compute the mean •a pre-computed mean vector	`nothing`

Notes:

- The output dimension `p` depends on both `maxoutdim` and `pratio`, as follows. Suppose the first `k` principal components preserve at least `pratio` of the total variance, while the first `k-1` preserves less than `pratio`, then the actual output dimension will be `min(k, maxoutdim)`.
- This function calls `pcacov` or `pcasvd` internally, depending on the choice of method.

Example:

```
using MultivariateStats

# suppose Xtr and Xte are training and testing data matrix,
# with each observation in a column

# train a PCA model
M = fit(PCA, Xtr; maxoutdim=100)

# apply PCA model to testing set
Yte = transform(M, Xte)

# reconstruct testing observations (approximately)
Xr = reconstruct(M, Yte)
```

Core Algorithms

Two algorithms are implemented in this package: `pcacov` and `pcastd`.

pcacov(*C*, *mean; ...*)

Compute PCA based on eigenvalue decomposition of a given covariance matrix `C`.

Parameters

- **C** – The covariance matrix.

- **mean** – The mean vector of original samples, which can be a vector of length `d`, or an empty vector `Float64[]` indicating a zero mean.

Returns The resultant PCA model.

Note This function accepts two keyword arguments: `maxoutdim` and `pratio`.

pcasvd(*Z*, *mean*, *tw; ...*)

Compute PCA based on singular value decomposition of a centralized sample matrix `Z`.

Parameters

- **Z** – provides centralized samples.
- **mean** – The mean vector of the **original** samples, which can be a vector of length `d`, or an empty vector `Float64[]` indicating a zero mean.

Returns The resultant PCA model.

Note This function accepts two keyword arguments: `maxoutdim` and `pratio`.

CHAPTER 4

Probabilistic Principal Component Analysis

Probabilistic Principal Component Analysis (PPCA) represents a constrained form of the Gaussian distribution in which the number of free parameters can be restricted while still allowing the model to capture the dominant correlations in a data set. It is expressed as the maximum likelihood solution of a probabilistic latent variable model *[BSHP06]*.

This package defines a `PPCA` type to represent a probabilistic PCA model, and provides a set of methods to access the properties.

Properties

Let `M` be an instance of `PPCA`, `d` be the dimension of observations, and `p` be the output dimension (*i.e* the dimension of the principal subspace)

`indim`(*M*)
: Get the input dimension `d`, *i.e* the dimension of the observation space.

`outdim`(*M*)
: Get the output dimension `p`, *i.e* the dimension of the principal subspace.

`mean`(*M*)
: Get the mean vector (of length `d`).

`projection`(*M*)
: Get the projection matrix (of size `(d, p)`). Each column of the projection matrix corresponds to a principal component.

  The principal components are arranged in descending order of the corresponding variances.

`loadings`(*M*)
: The factor loadings matrix (of size `(d, p)`).

`var`(*M*)
: The total residual variance.

Transformation and Construction

Given a probabilistic PCA model `M`, one can use it to transform observations into latent variables, as

$$\mathbf{z} = (\mathbf{W}^T\mathbf{W} + \sigma^2\mathbf{I})\mathbf{W}^T(\mathbf{x} - \boldsymbol{\mu})$$

or use it to reconstruct (approximately) the observations from latent variables, as

$$\tilde{\mathbf{x}} = \mathbf{W}\mathbb{E}[\mathbf{z}] + \boldsymbol{\mu}$$

Here, $\mathbf{W}$ is the factor loadings or weight matrix.

The package provides methods to do so:

`transform`(*M*, *x*)
: Transform observations `x` into latent variables.

  Here, `x` can be either a vector of length `d` or a matrix where each column is an observation.

`reconstruct`(*M*, *z*)
: Approximately reconstruct observations from the latent variable given in `z`.

  Here, `y` can be either a vector of length `p` or a matrix where each column gives the latent variables for an observation.

Data Analysis

One can use the `fit` method to perform PCA over a given dataset.

`fit`(*PPCA*, *X; ...*)
: Perform probabilistic PCA over the data given in a matrix `X`. Each column of `X` is an observation.

  This method returns an instance of `PCA`.

  Keyword arguments:

  Let `(d, n) = size(X)` be respectively the input dimension and the number of observations:

name	description	default
method	The choice of methods: •`:ml`: use maximum likelihood version of probabilistic PCA •`:em`: use EM verison of probabilistic PCA •`:bayes`: use Bayesian PCA	`:ml`
maxoutdim	Maximum output dimension.	`d-1`
mean	The mean vector, which can be either of: •`0`: the input data has already been centralized •`nothing`: this function will compute the mean •a pre-computed mean vector	`nothing`
tol	Convergence tolerance	`1.0e-6`
tot	Maximum number of iterations.	`1000`

Notes:

- This function calls `ppcaml`, `ppcaem` or `bayespca` internally, depending on the choice of method.

Example:

```
using MultivariateStats

# suppose Xtr and Xte are training and testing data matrix,
# with each observation in a column

# train a PCA model
M = fit(PPCA, Xtr; maxoutdim=100)

# apply PCA model to testing set
Yte = transform(M, Xte)

# reconstruct testing observations (approximately)
Xr = reconstruct(M, Yte)
```

Core Algorithms

Three algorithms are implemented in this package: `ppcaml`, `ppcaem`, and `bayespca`.

ppcaml (*Z*, *mean*, *tw; ...*)

Compute probabilistic PCA using on maximum likelihood formulation for a centralized sample matrix `Z`.

Parameters

- **Z** – provides centralized samples.
- **mean** – The mean vector of the **original** samples, which can be a vector of length `d`, or an empty vector `Float64[]` indicating a zero mean.

Returns The resultant PPCA model.

Note This function accepts two keyword arguments: `maxoutdim` and `tol`.

ppcaem(*C*, *mean; ...*)

Compute probabilistic PCA based on expectation-maximizaton algorithm for a given sample covariance matrix `S`.

Parameters

- **S** – The sample covariance matrix.
- **mean** – The mean vector of original samples, which can be a vector of length `d`, or an empty vector `Float64[]` indicating a zero mean.
- **n** – The number of observations.

Returns The resultant PPCA model.

Note This function accepts two keyword arguments: `maxoutdim`, `tol`, and `tot`.

bayespca(*C*, *mean; ...*)

Compute probabilistic PCA based on Bayesian algorithm for a given sample covariance matrix `S`.

Parameters

- **S** – The sample covariance matrix.
- **mean** – The mean vector of original samples, which can be a vector of length `d`, or an empty vector `Float64[]` indicating a zero mean.
- **n** – The number of observations.

Returns The resultant PPCA model.

Note This function accepts two keyword arguments: `maxoutdim`, `tol`, and `tot`.

Additional notes:

•Function uses the `maxoutdim` parameter as an upper boundary when it automatically determines the latent space dimensionality.

References

CHAPTER 5

Canonical Correlation Analysis

Canonical Correlation Analysis (CCA) is a statistical analysis technique to identify correlations between two sets of variables. Given two vector variables `X` and `Y`, it finds two projections, one for each, to transform them to a common space with maximum correlations.

The package defines a `CCA` type to represent a CCA model, and provides a set of methods to access the properties.

Properties

Let `M` be an instance of `CCA`, `dx` be the dimension of `X`, `dy` the dimension of `Y`, and `p` the output dimension (*i.e* the dimensio of the common space).

`xindim`(*M*)
: Get the dimension of `X`, the first set of variables.

`yindim`(*M*)
: Get the dimension of `Y`, the second set of variables.

`outdim`(*M*)
: Get the output dimension, *i.e* that of the common space.

`xmean`(*M*)
: Get the mean vector of `X` (of length `dx`).

`ymean`(*M*)
: Get the mean vector of `Y` (of length `dy`).

`xprojection`(*M*)
: Get the projection matrix for `X` (of size `(dx, p)`).

`yprojection`(*M*)
: Get the projection matrix for `Y` (of size `(dy, p)`).

`correlations`(*M*)
: The correlations of the projected componnents (a vector of length `p`).

Transformation

Given a CCA model, one can transform observations into both spaces into a common space, as

$$\mathbf{z}_x = \mathbf{P}_x^T(\mathbf{x} - \boldsymbol{\mu}_x)$$
$$\mathbf{z}_y = \mathbf{P}_y^T(\mathbf{y} - \boldsymbol{\mu}_y)$$

Here, $\mathbf{P}_x$ and $\mathbf{P}_y$ are projection matrices for `X` and `Y`; $\boldsymbol{\mu}_x$ and $\boldsymbol{\mu}_y$ are mean vectors.

This package provides methods to do so:

xtransform(*M*, *x*)
Transform observations in the X-space to the common space.

Here, `x` can be either a vector of length `dx` or a matrix where each column is an observation.

ytransform(*M*, *y*)
Transform observations in the Y-space to the common space.

Here, `y` can be either a vector of length `dy` or a matrix where each column is an observation.

Data Analysis

One can use the `fit` method to perform CCA over given datasets.

fit(*CCA*, *X*, *Y*; ...)
Perform CCA over the data given in matrices `X` and `Y`. Each column of `X` and `Y` is an observation.

`X` and `Y` should have the same number of columns (denoted by `n` below).

This method returns an instance of `CCA`.

Keyword arguments:

name	description	default
method	The choice of methods: •`:cov`: based on covariance matrices •`:svd`: based on SVD of the input data	`:svd`
outdim	The output dimension, *i.e* dimension of the common space	`min(dx, dy, n)`
mean	The mean vector, which can be either of: •`0`: the input data has already been centralized •`nothing`: this function will compute the mean •a pre-computed mean vector	`nothing`

Notes: This function calls `ccacov` or `ccasvd` internally, depending on the choice of method.

Core Algorithms

Two algorithms are implemented in this package: `pcacov` and `pcastd`.

ccacov(*Cxx*, *Cyy*, *Cxy*, *xmean*, *ymean*, *p*)
Compute CCA based on analysis of the given covariance matrices, using generalized eigenvalue decomposition.

Parameters

- **Cxx** – The covariance matrix of `X`.
- **Cyy** – The covariance matrix of `Y`.
- **Cxy** – The covariance matrix between `X` and `Y`.
- **xmean** – The mean vector of the original samples of `X`, which can be a vector of length `dx`, or an empty vector `Float64[]` indicating a zero mean.
- **ymean** – The mean vector of the original samples of `Y`, which can be a vector of length `dy`, or an empty vector `Float64[]` indicating a zero mean.
- **p** – The output dimension, *i.e* the dimension of the common space.

Returns The resultant CCA model.

ccasvd(*Zx*, *Zy*, *xmean*, *ymean*, *p*)
Compute CCA based on singular value decomposition of centralized sample matrices `Zx` and `Zy`.

Parameters

- **Zx** – The centralized sample matrix for `X`.
- **Zy** – The centralized sample matrix for `Y`.
- **xmean** – The mean vector of the **original** samples of `X`, which can be a vector of length `dx`, or an empty vector `Float64[]` indicating a zero mean.
- **ymean** – The mean vector of the **original** samples of `Y`, which can be a vector of length `dy`, or an empty vector `Float64[]` indicating a zero mean.
- **p** – The output dimension, *i.e* the dimension of the common space.

Returns The resultant CCA model.

CHAPTER 6

Classical Multidimensional Scaling

In general, Multidimensional Scaling (MDS) refers to techniques that transforms samples into lower dimensional space while preserving the inter-sample distances as well as possible.

Overview of Classical MDS

Classical MDS is a specific technique in this family that accomplishes the embedding in two steps:

1. Convert the distance matrix to a Gram matrix.

   This conversion is based on the following relations between a distance matrix `D` and a Gram matrix `G`:

   $$\mathrm{sqr}(\mathbf{D}) = \mathbf{g}\mathbf{1}^T + \mathbf{1}\mathbf{g}^T - 2\mathbf{G}$$

   Here, $\mathrm{sqr}(\mathbf{D})$ indicates the element-wise square of $\mathbf{D}$, and $\mathbf{g}$ is the diagonal elements of $\mathbf{G}$. This relation is itself based on the following decomposition of squared Euclidean distance:

   $$\|\mathbf{x} - \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^T\mathbf{y}$$

2. Perform eigenvalue decomposition of the Gram matrix to derive the coordinates.

Functions

This package provides functions related to classical MDS.

`gram2dmat`(*G*)
: Convert a Gram matrix `G` to a distance matrix.

`gram2dmat!(D, G)`
: Convert a Gram matrix `G` to a distance matrix, and write the results to `D`.

`dmat2gram`(*D*)
: Convert a distance matrix `D` to a Gram matrix.

dmat2gram!(G, D)
Convert a distance matrix `D` to a Gram matrix, and write the results to `G`.

classical_mds(*D*, *p*[, *dowarn=true*])
Perform classical MDS. This function derives a `p`-dimensional embedding based on a given distance matrix `D`.

It returns a coordinate matrix of size `(p, n)`, where each column is the coordinates for an observation.

Note: The Gramian derived from `D` may have nonpositive or degenerate eigenvalues. The subspace of nonpositive eigenvalues is projected out of the MDS solution so that the strain function is minimized in a least-squares sense. If the smallest remaining eigenvalue that is used for the MDS is degenerate, then the solution is not unique, as any linear combination of degenerate eigenvectors will also yield a MDS solution with the same strain value. By default, warnings are emitted if either situation is detected, which can be suppressed with `dowarn=false`.

If the MDS uses an eigenspace of dimension `m` less than `p`, then the MDS coordinates will be padded with `p-m` zeros each.

Reference:

```
@inbook{Borg2005,
Author = {Ingwer Borg and Patrick J. F. Groenen},
Title = {Modern Multidimensional Scaling: Theory and Applications},
Edition = {2},
Year = {2005},
Chapter = {12},
Doi = {10.1007/0-387-28981-X},
Pages = {201--268},
Series = {Springer Series in Statistics},
Publisher = {Springer},
}
```

CHAPTER 7

Linear Discriminant Analysis

Linear Discriminant Analysis are statistical analysis methods to find a linear combination of features for separating observations in two classes.

Note: Please refer to *Multi-class Linear Discriminant Analysis* for methods that can discriminate between multiple classes.

Overview of LDA

Suppose the samples in the positive and negative classes respectively with means: $\boldsymbol{\mu}_p$ and $\boldsymbol{\mu}_n$, and covariances $\mathbf{C}_p$ and $\mathbf{C}_n$. Then based on *Fisher's Linear Discriminant Criteria*, the optimal projection direction can be expressed as:

$$\mathbf{w} = \alpha \cdot (\mathbf{C}_p + \mathbf{C}_n)^{-1}(\boldsymbol{\mu}_p - \boldsymbol{\mu}_n)$$

Here α is an arbitrary non-negative coefficient.

Linear Discriminant

A linear discriminant functional can be written as

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

Here, `w` is the coefficient vector, and `b` is the bias constant.

This package uses the `LinearDiscriminant` type, defined as below, to capture a linear discriminant functional:

```
immutable LinearDiscriminant <: Discriminant
    w::Vector{Float64}
    b::Float64
end
```

This type comes with several methods. Let `f` be an instance of `LinearDiscriminant`

length (*f*)
Get the length of the coefficient vector.

evaluate (*f*, *x*)
Evaluate the linear discriminant value, *i.e* `w'x + b`.

When `x` is a vector, it returns a real value; when `x` is a matrix with samples in columns, it returns a vector of length `size(x, 2)`.

predict (*f*, *x*)
Make prediction. It returns `true` iff `evaluate(f, x)` is positive.

Data Analysis

The package provides several functions to perform Linear Discriminant Analysis.

ldacov (*Cp*, *Cn*, μp, μn)
Performs LDA given covariances and mean vectors.

Parameters

- **Cp** – The covariance matrix of the positive class.
- **Cn** – The covariance matrix of the negative class.
- μ**p** – The mean vector of the positive class.
- μ**n** – The mean vector of the negative class.

Returns The resultant linear discriminant functional of type `LinearDiscriminant`.

Note: The coefficient vector is scaled such that `w'`μ`p + b = 1` and `w'`μ`n + b = -1`.

ldacov (*C*, μp, μn)
Performs LDA given a covariance matrix and both mean vectors.

Parameters

- **C** – The pooled covariane matrix (*i.e* `(Cp + Cn)/2`)
- μ**p** – The mean vector of the positive class.
- μ**n** – The mean vector of the negative class.

Returns The resultant linear discriminant functional of type `LinearDiscriminant`.

Note: The coefficient vector is scaled such that `w'`μ`p + b = 1` and `w'`μ`n + b = -1`.

fit (*LinearDiscriminant*, *Xp*, *Xn*)
Performs LDA given both positive and negative samples.

Parameters

- **Xp** – The sample matrix of the positive class.
- **Xn** – The sample matrix of the negative class.

Returns The resultant linear discriminant functional of type `LinearDiscriminant`.

CHAPTER 8

Multi-class Linear Discriminant Analysis

Multi-class LDA is a generalization of standard two-class LDA that can handle arbitrary number of classes.

Overview

Multi-class LDA is based on the analysis of two scatter matrices: *within-class scatter matrix* and *between-class scatter matrix*.

Given a set of samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$, and their class labels $y_1, \ldots, y_n$:

The **within-class scatter matrix** is defined as:

$$\mathbf{S}_w = \sum_{i=1}^{n} (\mathbf{x}_i - \boldsymbol{\mu}_{y_i})(\mathbf{x}_i - \boldsymbol{\mu}_{y_i})^T$$

Here, $\boldsymbol{\mu}_k$ is the sample mean of the `k`-th class.

The **between-class scatter matrix** is defined as:

$$\mathbf{S}_b = \sum_{k=1}^{m} n_k (\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T$$

Here, `m` is the number of classes, $\boldsymbol{\mu}$ is the overall sample mean, and n_k is the number of samples in the `k`-th class.

Then, multi-class LDA can be formulated as an optimization problem to find a set of linear combinations (with coefficients $\mathbf{w}$) that maximizes the ratio of the between-class scattering to the within-class scattering, as

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmax}} \frac{\mathbf{w}^T \mathbf{S}_b \mathbf{w}}{\mathbf{w}^T \mathbf{S}_w \mathbf{w}}$$

The solution is given by the following generalized eigenvalue problem:

$$\mathbf{S}_b \mathbf{w} = \lambda \mathbf{S}_w \mathbf{w} \tag{8.1}$$

Generally, at most `m - 1` generalized eigenvectors are useful to discriminate between `m` classes.

When the dimensionality is high, it may not be feasible to construct the scatter matrices explicitly. In such cases, see *SubspaceLDA* below.

Normalization by number of observations

An alternative definition of the within- and between-class scatter matrices normalizes for the number of observations in each group:

$$\mathbf{S}_w^* = n \sum_{k=1}^{m} \frac{1}{n_k} \sum_{i|y_i=k} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T$$

$$\mathbf{S}_b^* = n \sum_{k=1}^{m} (\boldsymbol{\mu}_k - \boldsymbol{\mu}^*)(\boldsymbol{\mu}_k - \boldsymbol{\mu}^*)^T$$

where

$$\boldsymbol{\mu}^* = \frac{1}{k} \sum_{k=1}^{m} \boldsymbol{\mu}_k.$$

This definition can sometimes be more useful when looking for directions which discriminate among clusters containing widely-varying numbers of observations.

Multi-class LDA

The package defines a `MulticlassLDA` type to represent a multi-class LDA model, as:

```
type MulticlassLDA
    proj::Matrix{Float64}
    pmeans::Matrix{Float64}
    stats::MulticlassLDAStats
end
```

Here, `proj` is the projection matrix, `pmeans` is the projected means of all classes, `stats` is an instance of `MulticlassLDAStats` that captures all statistics computed to train the model (which we will discuss later).

Several methods are provided to access properties of the LDA model. Let `M` be an instance of `MulticlassLDA`:

`indim`(*M*)
: Get the input dimension (*i.e* the dimension of the observation space).

`outdim`(*M*)
: Get the output dimension (*i.e* the dimension of the transformed features).

`projection`(*M*)
: Get the projection matrix (of size `d x p`).

`mean`(*M*)
: Get the overall sample mean vector (of length `d`).

`classmeans`(*M*)
: Get the matrix comprised of class-specific means as columns (of size `(d, m)`).

`classweights`(*M*)
: Get the weights of individual classes (a vector of length `m`). If the samples are not weighted, the weight equals the number of samples of each class.

`withinclass_scatter`(*M*)
: Get the within-class scatter matrix (of size `(d, d)`).

`betweenclass_scatter`(*M*)
: Get the between-class scatter matrix (of size `(d, d)`).

transform(*M*, *x*)

Transform input sample(s) in `x` to the output space. Here, `x` can be either a sample vector or a matrix comprised of samples in columns.

In the pratice of classification, one can transform testing samples using this `transform` method, and compare them with `M.pmeans`.

Data Analysis

One can use `fit` to perform multi-class LDA over a set of data:

fit (*MulticlassLDA*, *nc*, *X*, *y; ...*)

Perform multi-class LDA over a given data set.

Parameters

- **nc** – the number of classes
- **X** – the matrix of input samples, of size `(d, n)`. Each column in `X` is an observation.
- **y** – the vector of class labels, of length `n`. Each element of `y` must be an integer between `1` and `nc`.

Returns The resultant multi-class LDA model, of type `MulticlassLDA`.

Keyword arguments:

name	description	default
method	The choice of methods: •`:gevd`: based on generalized eigenvalue decomposition •`:whiten`: first derive a whitening transform from `Sw` and then solve the problem based on eigenvalue decomposition of the whiten `Sb`.	`:gevd`
outdim	The output dimension, *i.e* dimension of the transformed space	`min(d, nc-1)`
regcoef	The regularization coefficient. A positive value `regcoef * eigmax(Sw)` is added to the diagonal of `Sw` to improve numerical stability.	`1.0e-6`

Note: The resultant projection matrix `P` satisfies:

$$\mathbf{P}^T(\mathbf{S}_w + \kappa\mathbf{I})\mathbf{P} = \mathbf{I}$$

Here, κ equals `regcoef * eigmax(Sw)`. The columns of `P` are arranged in descending order of the corresponding generalized eigenvalues.

Note that `MulticlassLDA` does not currently support the normalized version using $\mathbf{S}_w^*$ and $\mathbf{S}_b^*$ (see `SubspaceLDA` below).

Task Functions

The multi-class LDA consists of several steps:

1. Compute statistics, such as class means, scatter matrices, etc.
2. Solve the projection matrix.
3. Construct the model.

Sometimes, it is useful to only perform one of these tasks. The package exposes several functions for this purpose:

multiclass_lda_stats(*nc*, *X*, *y*)
Compute statistics required to train a multi-class LDA.

Parameters

- **nc** – the number of classes
- **X** – the matrix of input samples.
- **y** – the vector of class labels.

This function returns an instance of `MulticlassLDAStats`, defined as below, that captures all relevant statistics.

```
type MulticlassLDAStats
    dim::Int                    # sample dimensions
    nclasses::Int               # number of classes
    cweights::Vector{Float64}   # class weights
    tweight::Float64            # total sample weight
    mean::Vector{Float64}       # overall sample mean
    cmeans::Matrix{Float64}     # class-specific means
    Sw::Matrix{Float64}         # within-class scatter matrix
    Sb::Matrix{Float64}         # between-class scatter matrix
end
```

This type has the following constructor. Under certain circumstances, one might collect statistics in other ways and want to directly construct this instance.

MulticlassLDAStats(*cweights*, *mean*, *cmeans*, *Sw*, *Sb*)
Construct an instance of type `MulticlassLDAStats`.

Parameters

- **cweights** – the class weights, a vector of length `m`.
- **mean** – the overall sample mean, a vector of length `d`.
- **cmeans** – the class-specific sample means, a matrix of size `(d, m)`.
- **Sw** – the within-class scatter matrix, a matrix of size `(d, d)`.
- **Sb** – the between-class scatter matrix, a matrix of size `(d, d)`.

multiclass_lda(*S; ...*)
Perform multi-class LDA based on given statistics. Here `S` is an instance of `MulticlassLDAStats`.

This function accepts the following keyword arguments (as above): `method`, `outdim`, and `regcoef`.

mclda_solve(*Sb*, *Sw*, *method*, *p*, *regcoef*)
Solve the projection matrix given both scatter matrices.

Parameters

- **Sb** – the between-class scatter matrix.
- **Sw** – the within-class scatter matrix.
- **method** – the choice of method, which can be either `:gevd` or `:whiten`.
- **p** – output dimension.
- **regcoef** – regularization coefficient.

mclda_solve!(Sb, Sw, method, p, regcoef)

Solve the projection matrix given both scatter matrices.

Note: In this function, `Sb` and `Sw` will be overwritten (saving some space).

Subspace LDA

The package also defines a `SubspaceLDA` type to represent a multi-class LDA model for high-dimensional spaces. `MulticlassLDA`, because it stores the scatter matrices, is not well-suited for high-dimensional data. For example, if you are performing LDA on images, and each image has `10^6` pixels, then the scatter matrices would contain `10^12` elements, far too many to store directly. `SubspaceLDA` calculates the projection direction without the intermediary of the scatter matrices, by focusing on the subspace that lies within the span of the within-class scatter. This also serves to regularize the computation.

```
immutable SubspaceLDA{T<:AbstractFloat}
    projw::Matrix{T}   # P, project down to the subspace spanned by within-class␣
↪scatter
    projLDA::Matrix{T} # L, LDA directions in the projected subspace
    λ::Vector{T}
    cmeans::Matrix{T}
    cweights::Vector{Int}
end
```

This supports all the same methods as `MulticlassLDA`, with the exception of the functions that return a scatter matrix. The overall projection is represented as a factorization `P*L`, where `P'*x` projects data points to the subspace spanned by the within-class scatter, and `L` is the LDA projection in the subspace. The projection directions `w` (the columns of `projection(M)`) satisfy the equation

$$\mathbf{P}^T\mathbf{S}_b\mathbf{w} = \lambda\mathbf{P}^T\mathbf{S}_w\mathbf{w}.$$

When `P` is of full rank (e.g., if there are more data points than dimensions), then this equation guarantees that Eq. (8.1) will also hold.

SubspaceLDA also supports the normalized version of LDA via the `normalize` keyword:

```
M = fit(SubspaceLDA, X, label; normalize=true)
```

would perform LDA using the equivalent of $\mathbf{S}_w^*$ and $\mathbf{S}_b^*$.

CHAPTER 9

Independent Component Analysis

Independent Component Analysis (ICA) is a computational technique for separating a multivariate signal into additive subcomponents, with the assumption that the subcomponents are non-Gaussian and independent from each other.

There are multiple algorithms for ICA. Currently, this package implements the Fast ICA algorithm.

ICA

The package uses a type `ICA`, defined below, to represent an ICA model:

```
type ICA
    mean::Vector{Float64}    # mean vector, of length m (or empty to indicate zero␣
↪mean)
    W::Matrix{Float64}       # component coefficient matrix, of size (m, k)
end
```

Note: Each column of `W` here corresponds to an independent component.

Several methods are provided to work with `ICA`. Let `M` be an instance of `ICA`:

indim(*M*)
: Get the input dimension, *i.e* the number of observed mixtures.

outdim(*M*)
: Get the output dimension, *i.e* the number of independent components.

mean(*M*)
: Get the mean vector.

  Note: if `M.mean` is empty, this function returns a zero vector of length `indim(M)`.

transform(*M*, *x*)
: Transform `x` to the output space to extract independent components, as $\mathbf{W}^T(\mathbf{x} - \boldsymbol{\mu})$.

Data Analysis

One can use `fit` to perform ICA over a given data set.

fit (*ICA*, *X*, *k; ...*)
Perform ICA over the data set given in `X`.

Parameters

- **X** – The data matrix, of size `(m, n)`. Each row corresponds to a mixed signal, while each column corresponds to an observation (*e.g* all signal value at a particular time step).
- **k** – The number of independent components to recover.

Returns

The resultant ICA model, an instance of type `ICA`.

Note: If `do_whiten` is `true`, the return `W` satisfies $\mathbf{W}^T\mathbf{C}\mathbf{W} = \mathbf{I}$, otherwise `W` is orthonormal, *i.e* $\mathbf{W}^T\mathbf{W} = \mathbf{I}$

Keyword Arguments:

name	description	default
alg	The choice of algorithm (must be `:fastica`)	`:fastica`
fun	The approx neg-entropy functor. It can be obtained using the function `icagfun`. Now, it accepts the following values: •`icagfun(:tanh)` •`icagfun(:tanh, a)` •`icagfun(:gaus)`	`icagfun(:tanh)`
do_whiten	Whether to perform pre-whitening	`true`
maxiter	Maximum number of iterations	`100`
tol	Tolerable change of `W` at convergence	`1.0e-6`
mean	The mean vector, which can be either of: •`0`: the input data has already been centralized •`nothing`: this function will compute the mean •a pre-computed mean vector	`nothing`
winit	Initial guess of `W`, which should be either of: •empty matrix: the function will perform random initialization •a matrix of size `(k, k)` (when `do_whiten`) •a matrix of size `(m, k)` (when `!do_whiten`)	`zeros(0,0)`
verbose	Whether to display iteration information	`false`

Core Algorithms

The package also exports functions of the core algorithms. Sometimes, it can be more efficient to directly invoke them instead of going through the `fit` interface.

`fastica!(W, X, fun, maxiter, tol, verbose)`

Invoke the Fast ICA algorithm.

Parameters

- **`W`** – The initial un-mixing matrix, of size `(m, k)`. The function updates this matrix inplace.
- **`X`** – The data matrix, of size `(m, n)`. This matrix is input only, and won't be modified.
- **`fun`** – The approximate neg-entropy functor, which can be obtained using `icagfun` (see above).
- **`maxiter`** – Maximum number of iterations.

- **tol** – Tolerable change of W at convergence.
- **verbose** – Whether to display iteration information.

Returns The updated W.

Note: The number of components is inferred from W as `size(W, 2)`.

Notes:

All methods implemented in this package adopt the column-major convention of JuliaStats: in a data matrix, each column corresponds to a sample/observation, while each row corresponds to a feature (variable or attribute).

Bibliography

[BSHP06] Bishop, C. M. Pattern Recognition and Machine Learning, 2006.

Index

Y