Machine Learning Kernels Documentation

Release 0.0.3

Tim Thatcher

Oct 06, 2017

Contents

MLKernels.jl is a Julia package for Mercer kernel functions (or the covariance functions of Gaussian processes) that are used in the kernel methods of machine learning. This package provides a collection of kernel datatypes for representing kernel functions as well as an efficient set of methods to compute or approximate kernel matrices. The package has no dependencies beyond base Julia.

CHAPTER 1

Installation

The package may be added by running one of the following lines of code:

```
# Latest stable release in Metadata:
Pkg.add("MLKernels")

# Most up-to-date (not stable):
Pkg.checkout("MLKernels")

# Development (bleeding edge):
Pkg.checkout("MLKernels", "dev")
```

CHAPTER 2

Getting Started

Documentation on the interface implemented by MLKernels.jl is available under the interface section listed in the table of contents below or on the sidebar:

Interface

Storage

MLKernels.jl allows for data matrices to be stored in one of two ways with respect to the observations based on parameters provided by the user. In order to specify the ordering used, a subtype of the `MemoryLayout` abstract type can be provided as a parameter to any methods taking matrices as a parameter:

type `RowMajor`

Identifies when each observation vector corresponds to a row in the data matrix. This is commonly used in the field of statistics in the context of design matrices. For example, for data matrix $\mathbf{X}$ consisting of observations $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$:

$$\mathbf{X}_{row} = \begin{bmatrix} \leftarrow \mathbf{x}_1 \rightarrow \\ \leftarrow \mathbf{x}_2 \rightarrow \\ \vdots \\ \leftarrow \mathbf{x}_n \rightarrow \end{bmatrix}$$

When row-major ordering is used, then the kernel matrix of $\mathbf{X}$ will match the dimensions of $\mathbf{X}^\intercal\mathbf{X}$. Similarly, the kernel matrix will match the dimension of $\mathbf{X}^\intercal\mathbf{Y}$ for row-major ordering of data matrix $\mathbf{X}$ and $\mathbf{Y}$.

type `ColumnMajor`

Identifies when each observation vector corresponds to the column of the data matrix. This is much more common in Machine Learning communities:

$$\mathbf{X}_{col} = \mathbf{X}_{row}^\intercal = \begin{bmatrix} \uparrow & \uparrow & & \uparrow \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x_n} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}$$

With column-major ordering, the kernel matrix will match the dimensions of $\mathbf{X}\mathbf{X}^\intercal$. Similarly, the kernel matrix of data matrices $\mathbf{X}$ and $\mathbf{Y}$ match the dimensions of $\mathbf{X}\mathbf{Y}^\intercal$.

Note: Row-major and column-major ordering in this context do not refer to the physical storage ordering of the underlying matrices (in the case of Julia, all arrays are in column-major ordering). These properties refer to the ordering of observations within a data matrix; either per-row or per-column.

Essentials

The primary feature of the `MLKernels` package is the ability to efficiently compute kernel functions and kernel matrices. The interface is outlined below:

`ismercer`(*κ::Kernel*) → Bool
: Returns `true` if kernel κ is a Mercer kernel; `false` otherwise.

`isnegdef`(*κ::Kernel*) → Bool
: Returns `true` if the kernel κ is a negative definite kernel; `false` otherwise.

`isisotropic`(*κ::Kernel*) → Bool
: Returns `true` if the kernel κ is an isotropic kernel; `false` otherwise.

`isstationary`(*κ::Kernel*) → Bool
: Returns `true` if the kernel κ is a stationary kernel; `false` otherwise.

`kernel`(*κ::Kernel*, *x*, *y*)
: Apply the kernel κ to `x` and `y` where `x` and `y` are vectors or scalars of some subtype of `Real`.

`kernelmatrix`([*σ::MemoryLayout*], *κ::Kernel*, *X::Matrix*[, *symmetrize::Bool*])
: Calculate the kernel matrix of `X` with respect to kernel κ.

  See the *storage notes* to determine the value of σ; by default σ is set to `RowMajor()`. Set `symmetrize` to `false` to fill only the upper triangle of `K`, otherwise the upper triangle will be copied to the lower triangle.

`kernelmatrix!`(*P::Matrix*, *σ::MemoryLayout*, *κ::Kernel*, *X::Matrix*, *symmetrize::Bool*)
: In-place version of `kernelmatrix` where pre-allocated matrix `K` will be overwritten with the kernel matrix.

`kernelmatrix`([*σ::MemoryLayout*], *κ::Kernel*, *X::Matrix*, *Y::Matrix*)
: Calculate the pairwise matrix of `X` and `Y` with respect to kernel κ.

  See the *storage notes* to determine the value of σ. By default σ is set to `RowMajor()`.

`kernelmatrix!`(*K::Matrix*, *σ::MemoryLayout*, *κ::Kernel*, *X::Matrix*, *Y::Matrix*)
: In-place version of `kernelmatrix` where pre-allocated matrix `K` will be overwritten with the kernel matrix.

`centerkernelmatrix`(*K::Matrix*)
: Centers the (rectangular) kernel matrix `K` with respect to the implicit Kernel Hilbert Space according to the following formula:

$$[\mathbf{K}]_{ij} = \langle \phi(\mathbf{x}_i) - \mu_{\phi\mathbf{x}}, \phi(\mathbf{y}_j) - \mu_{\phi\mathbf{y}} \rangle$$

  Where $\mu_{\phi\mathbf{x}}$ and $\mu_{\phi\mathbf{x}}$ are given by:

$$\mu_{\phi\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n}\phi(\mathbf{x}_i) \qquad \mu_{\phi\mathbf{y}} = \frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{y}_i)$$

`centerkernelmatrix!`(*K::Matrix*)
: In-place version of `centerkernelmatrix` overwriting `K`.

Approximation

In many cases, a fast, approximate results is more important than a perfect result. The Nystrom method can be used to generate a factorization that can be used to approximate a large, symmetric kernel matrix. Given data matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$ (one observation per row) and kernel matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$, the Nystrom method takes a sample S of the observations of $\mathbf{X}$ of size $s < n$ and generates a factorization such that:

$$\mathbf{K} \approx \mathbf{C}^{\intercal}\mathbf{W}\mathbf{C}$$

Where $\mathbf{W}$ is the $s \times s$ pseudo-inverse of the sample kernel matrix based on S and $\mathbf{C}$ is a $s \times n$ matrix.

The Nystrom method uses an eigendecomposition of the sample kernel matrix of $\mathbf{X}$ to estimate $\mathbf{K}$. Generally, the order of $\mathbf{K}$ must be quite large and the sampling ratio small (ex. 15% or less) for the cost of the computing the full kernel matrix to exceed that of the eigendecomposition. This method will be more effective for kernels that are not a direct function of the dot product as they are not able to make use of BLAS in computing the full matrix $\mathbf{K}$ and the cross-over point will occur for smaller $\mathbf{K}$.

MLKernels.jl implements the Nystrom approximation:

type **`NystromFact`**

Type for storing a Nystrom factorization. The factorization contains two fields: `W` and `C` as described above.

`nystrom`(*σ::MemoryLayout*, *κ::Kernel*, *X::Matrix*, *S::Vector*) → NystromFact

Computes a factorization of Nystrom approximation of the square kernel matrix of data matrix `X` with respect to kernel κ. Returns type `NystromFact` which stores a Nystrom factorization:

`kernelmatrix`(*CtWC::NystromFact]*)

Computes the approximate kernel matrix using the Nystrom factorization.

Pairwise Functions

The `PairwiseFunctions` submodule is provided to compute symmetric real-valued functions (pairwise functions) of the form:

$$f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$

Each kernel function has an underlying pairwise function. Similar to kernel functions, a pairwise function can be evaluated for each pair of observations across two data matrices to produce a matrix of pairwise evaluations. Given data matrix $\mathbf{X}$, data matrix $\mathbf{Y}$ and pairwise function f, the pairwise matrix $\mathbf{P}$ is defined by:

$$\mathbf{P} = [f(\mathbf{x}_i, \mathbf{y}_j)]_{ij} \qquad \forall \mathbf{x}_i \in \mathbf{X}, \quad \mathbf{y}_j \in \mathbf{Y}$$

The interface is outlined below:

`pairwise`(*f::PairwiseFunction*, *x*, *y*)

Apply the function `f` to `x` and `y` where `x` and `y` are vectors or scalars of some subtype of `Real`.

`pairwisematrix`([*σ::MemoryLayout*], *f::Kernel*, *X::Matrix*[, *symmetrize::Bool*])

Calculate the pairwise matrix of `X` with respect to function `f`.

See the *storage notes* to determine the value of σ; by default σ is set to `RowMajor()`. Set `symmetrize` to `false` to fill only the upper triangle of `P`, otherwise the upper triangle will be copied to the lower triangle.

`pairwisematrix!`(*P::Matrix*, *σ::MemoryLayout*, *f::PairwiseFunction*, *X::Matrix*, *symmetrize::Bool*)

In-place version of `pairwisematrix` where pre-allocated matrix `P` will be overwritten with the pairwise matrix.

pairwisematrix([*σ::MemoryLayout*], *f::PairwiseFunction*, *X::Matrix*, *Y::Matrix*)
Calculate the pairwise matrix of `X` and `Y` with respect to function `f`.

See the *storage notes* to determine the value of σ. By default σ is set to `RowMajor()`.

pairwisematrix!(*P::Matrix*, *σ::MemoryLayout*, *f::PairwiseFunction*, *X::Matrix*, *Y::Matrix*)
In-place version of `kernelmatrix` where pre-allocated matrix `K` will be overwritten with the kernel matrix.

Hyper Parameters

The submodule `HyperParameters` defines a `HyperParameter` type as well as a `Bound` and `Interval` type. The hyper parameter type stores the current value of a hyper parameter as well as an interval that defines the domain of the hyper parameter. Each `Kernel` type is a struct of `HyperParameter` instances.

Often, hyper parameter values are restricted to an interval with an open bounded start point or end point (ex. $\alpha > 0$). Exclusive finite endpoints such as these are often disallowed in optimization algorithms. This module includes two transformations to work around these constraints:

- `theta`: The function θ is used to transform a parameter restricted to a finite open-bounded interval to an interval without finite open bounds.
- `eta`: The function η is the inverse of θ. It converts from values in the transformed space back to the original parameter space.

The specific form of θ and η depends on the interval that the parameter is restricted to. Given finite a, finite b and parameter α, functions θ and η are defined as follows:

Domain α	Function $\theta_\alpha = \theta(\alpha)$	Domain θ_α	Function $\eta(\theta_\alpha)$
(a, b)	$\log(\alpha - a) - \log(b - \alpha)$	$(-\infty, \infty)$	$(b\exp(\theta_\alpha) + a)/(1 + \exp(\theta_\alpha))$
$(a, b]$	$\log(\alpha - a)$	$(-\infty, \log(b - a)]$	$\exp(\theta_\alpha) + a$
$[a, b)$	$\log(b - \alpha)$	$(-\infty, \log(b - a)]$	$b - \exp(\theta_\alpha)$
(a, ∞)	$\log(\alpha - a)$	$(-\infty, \infty)$	$\exp(\theta_\alpha) + a$
$(-\infty, b)$	$\log(b - \alpha)$	$(-\infty, \infty)$	$b - \exp(\theta_\alpha)$
$(-\infty, \infty)$	N/A	N/A	N/A
$[a, b]$	N/A	N/A	N/A
$(-\infty, b]$	N/A	N/A	N/A
$[a, \infty)$	N/A	N/A	N/A

The following functions are supported by the hyper parameter submodule:

ClosedBound(*a::Real*) → ClosedBound
Constructs a `ClosedBound` type which is used to signify a closed bound on an interval.

OpenBound(*a::Real*) → OpenBound
Constructs an `OpenBound` type which is used to signify an open bound on an interval. Type `T` must not be integer - only closed bounds are used for integers.

NullBound(*a::DataType*) → NullBound
Constructs a `NullBound` type which is used to signify an infinite open bound on an interval.

Interval(*a::Bound*, *b::Bound*) → Interval
Constructs an `Interval` type. The interval type is used to represent box constraints on parameters. This can be used to restrict the values a hyper parameter may take on.

The `Interval` type is also used to define the form of `theta`.

interval(*a::Union{Bound, Void}*, *b::Union{Bound, Void}*) → Interval
Constructs an `Interval` type. If `nothing` is provided for `a` or `b`, then a `NullBound` will be substituted. If both `a` and `b` are `nothing`, the interval defaults to an unbounded `Interval{Float64}` type.

HyperParameter(*a::Real*, *I::Interval*) → HyperParameter
Constructs a hyper parameter with value `a` and domain restriction `I`. If `a` is an invalid value for `I`, then the constructor will fail.

checkvalue(*P::HyperParameter*, *x::Real*)
Checks if `x` falls within the hyper parameter domain of `P`.

getvalue(*P::HyperParameter*)
Gets the current value of hyper parameter `P`.

setvalue!(*P::HyperParameter*, *x::Real*)
Sets the value of `P` to `x`.

checktheta(*P::HyperParameter*, *x::Real*)
Checks if $\eta(x)$ falls within the hyper parameter domain of `P`.

gettheta(*P::HyperParameter*)
Gets the current value of $\theta(P)$ of hyper parameter `P`.

settheta!(*P::HyperParameter*, *x::Real*)
Sets the value of `P` to $\eta(x)$.

Kernel Functions

Exponential Kernel

ExponentialKernel([α::Real=1]) <: MercerKernel
The exponential kernel is given by the formula:

$$\kappa(\mathbf{x}, \mathbf{y}) = \exp\left(-\alpha||\mathbf{x} - \mathbf{y}||\right) \qquad \alpha > 0$$

where α is a scaling parameter of the Euclidean distance. The exponential kernel, also known as the Laplacian kernel, is an isotropic Mercer kernel. The constructor is aliased by `LaplacianKernel`, so both names may be used:

```
ExponentialKernel()   # Default is Float64 with α = 1.0
LaplacianKernel(1)    # Integers will be converted to Float64
```

Squared Exponential Kernel

SquaredExponentialKernel([α::Real=1]) <: MercerKernel
The squared exponential kernel, or alternatively the Gaussian kernel, is identical to the exponential kernel except that the Euclidean distance is squared:

$$\kappa(\mathbf{x}, \mathbf{y}) = \exp\left(-\alpha||\mathbf{x} - \mathbf{y}||^2\right) \qquad \alpha > 0$$

where α is a scaling parameter of the squared Euclidean distance. Just like the exponential kernel, the squared exponential kernel is an isotropic Mercer kernel. The squared exponential kernel is more commonly known as the radial basis kernel within machine learning communities. All aliases may be used in MLKernels.jl:

```
GaussianKernel()
RadialBasisKernel()
SquaredExponentialKernel()
```

Gamma Exponential Kernel

GammaExponentialKernel([α::Real=1 [,γ::Real=1]]) <: MercerKernel

The gamma exponential kernel is a generalization of the exponential and squared exponential kernels:

$$\kappa(\mathbf{x},\mathbf{y}) = \exp\left(-\alpha||\mathbf{x}-\mathbf{y}||^{\gamma}\right) \qquad \alpha > 0,\ 0 < \gamma \leq 1$$

where α is a scaling parameter and γ is a shape parameter.

Rational-Quadratic Kernel

RationalQuadraticKernel([α::Real=1 [,β::Real=1]]) <: MercerKernel

The rational-quadratic kernel is given by:

$$\kappa(\mathbf{x},\mathbf{y}) = \left(1+\alpha||\mathbf{x},\mathbf{y}||^{2}\right)^{-\beta} \qquad \alpha > 0,\ \beta > 0$$

where α is a scaling parameter and β is a shape parameter. This kernel can be seen as an infinite sum of Gaussian kernels. If one sets $\alpha = \alpha_0/\beta$, then taking the limit $\beta \to \infty$ results in the Gaussian kernel with scaling parameter α_0.

Gamma-Rational Kernel

RationalQuadraticClass([α::Real [,β::Real [,γ::Real]]]) <: MercerKernel

The gamma-rational kernel is a generalization of the rational-quadratic kernel with an additional shape parameter:

$$\kappa(\mathbf{x},\mathbf{y}) = (1+\alpha||\mathbf{x},\mathbf{y}||^{\gamma})^{-\beta} \qquad \alpha > 0,\ \beta > 0,\ 0 < \gamma \leq 1$$

where α is a scaling parameter and β and γ are shape parameters.

Matern Kernel

MaternKernel([ν::Real=1 [,θ::Real=1]]) <: MercerKernel

The Matern kernel is a **Mercer** kernel *[ras]* given by:

$$\kappa(\mathbf{x},\mathbf{y}) = \frac{1}{2^{\nu-1}\Gamma(\nu)}\left(\frac{2\sqrt{\nu}||\mathbf{x}-\mathbf{y}||}{\theta}\right)^{\nu} K_{\nu}\left(\frac{2\sqrt{\nu}||\mathbf{x}-\mathbf{y}||}{\theta}\right)$$

where Γ is the gamma function, K_ν is the modified Bessel function of the second kind, $\nu > 0$ and $\theta > 0$.

Linear Kernel

LinearKernel([a::Real=1 [,c::Real=1]]) <: MercerKernel

The linear kernel is a **Mercer** kernel given by:

$$\kappa(\mathbf{x},\mathbf{y}) = a\mathbf{x}^{\intercal}\mathbf{y} + c \qquad \alpha > 0,\ c \geq 0$$

Polynomial Kernel

`PolynomialKernel([a::Real=1 [,c::Real=1 [,d::Integer=3]]]) <: MercerKernel`

The polynomial kernel is a **Mercer** kernel given by:

$$\kappa(\mathbf{x},\mathbf{y}) = (a\mathbf{x}^{\intercal}\mathbf{y} + c)^d \qquad \alpha > 0,\ c \geq 0,\ d \in \mathbb{Z}_{+}$$

Exponentiated Kernel

`ExponentiatedKernel([a::Real=1]) <: MercerKernel`

The exponentiated kernel is a **Mercer** kernel given by:

$$\kappa(\mathbf{x},\mathbf{y}) = \exp\left(a\mathbf{x}^{\intercal}\mathbf{y}\right) \qquad a > 0$$

Periodic Kernel

`PeriodicKernel([`α`::Real=1 [,p::Real=`π`]]) <: MercerKernel`

The periodic kernel is given by:

$$\kappa(\mathbf{x},\mathbf{y}) = \exp\left(-\alpha \sum_{i=1}^{n} \sin(p(x_i - y_i))^2\right) \qquad p > 0,\ \alpha > 0$$

where $\mathbf{x}$ and $\mathbf{y}$ are n dimensional vectors. The parameters p and α are scaling parameters for the periodicity and the magnitude, respectively. This kernel is useful when data has periodicity to it.

Sigmoid Kernel

`SigmoidKernel([a::Real=1 [,c::Real=1]]) <: Kernel`

The sigmoid kernel is given by:

$$\kappa(\mathbf{x},\mathbf{y}) = \tanh(a\mathbf{x}^{\intercal}\mathbf{y} + c) \qquad \alpha > 0,\ c \geq 0$$

The sigmoid kernel is a not a true kernel, although it has been used in application.

Kernel Theory

The Kernel Trick

Many machine and statistical learning algorithms, such as support vector machines and principal components analysis, are based on **inner products**. These methods can often be generalized through use of the **kernel trick** to create a nonlinear decision boundary without using an explicit mapping to another space.

The kernel trick makes use of **Mercer kernels** which operate on vectors in the input space but can be expressed as inner products in another space. In other words, if $\mathcal{X}$ is the input vector space and κ is the Mercer kernel function, then for some vector space $\mathcal{V}$ there exists a function ϕ such that:

$$\kappa(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle_{\mathcal{V}} \qquad x_1, x_2 \in \mathcal{X}$$

In machine learning, the vector space $\mathcal{X}$ is known as the feature space and the function ϕ is known as a feature map. A simple example of a feature map can be shown with the Polynomial Kernel:

$$\kappa(\mathbf{x},\mathbf{y}) = (a\mathbf{x}^{\intercal}\mathbf{y} + c)^d \qquad \mathbf{x},\mathbf{y} \in \mathbb{R}^n, \quad a, c \in \mathbb{R}_{+} \quad d \in \mathbb{Z}_{+}$$

In our example, we will use $n = 2$, $d = 2$, $a = 1$ and $c = 0$. Substituting these values in, we get the following kernel function:

$$\kappa(\mathbf{x}, \mathbf{y}) = (x_1 y_1 + x_2 y_2)^2 = x_1^2 y_1^2 + x_1 x_2 y_1 y_2 + x_2^2 y_2^2 = \phi(\mathbf{x})^\intercal \phi(\mathbf{y})$$

Where the feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ is defined by:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$$

The advantage of the implicit feature map is that we may transform non-linearly data into linearly separable data in the implicit space. For example, suppose we have a single feature and two classes that cannot be separated using a linear function of that feature:

Using the feature map above, we create a data set that is linearly separable:

Kernels

The kernel methods are a class of algorithms that are used for pattern analysis. These methods make use of **kernel** functions. A symmetric, real valued kernel function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is said to be **positive definite** or **Mercer** if and only:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \geq 0$$

for all $n \in \mathbb{N}$, $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subseteq \mathcal{X}$ and $\{c_1, \ldots, c_n\} \subseteq \mathbb{R}$. Similarly, a real valued kernel function is said to be **negative definite** if and only if:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \leq 0 \qquad \sum_{i=1}^{n} c_i = 0$$

for $n \geq 2$, $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subseteq \mathcal{X}$ and $\{c_1, \ldots, c_n\} \subseteq \mathbb{R}$. In machine learning literature, **conditionally positive definite** kernels are often studied instead. This is simply a reversal of the above inequality. Trivially, every negative definite kernel can be transformed into a conditionally positive definite kernel by negation.

Nystrom method

References

A listing of the implemented kernel functions and their properties is available on the kernels page and documentation on the theory surrounding kernel functions and the kernel trick is available in the kernel theory section.

Bibliography

[berg] Berg C, Christensen JPR, Ressel P. 1984. *Harmonic Analysis on Semigroups*. Springer-Verlag New York. Chapter 3, General Results on Positive and Negative Definite Matrices and Kernels; p. 66-85.

[bou] Bouboulis P. 2014. *Academic Press Library in Signal Processing, Volume 1: Array and Statistical Signal Processing (1st ed.)*. Academic Press. Chapter 17, Online Learning in Reproducing Kernel Hilbert Spaces; p. 883-987.

[gen] Genton M.G. 2002. *Classes of kernels for machine learning: a statistics perspective*. The Journal of Machine Learning Research. Volume 2 (March 2002), 299-312.

[ras] Rasmussen C, Williams CKI. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press. Chapter 4, Covariance Functions; p. 79-104.

Index