Online and scalable kernel methods

I. Santamaría, S. Van Vaerenbergh

GTAS, Universidad de Cantabria

14 de febrero de 2022

Maśter Universitario Oficial **Data Science**

con el apoyo del

Contents

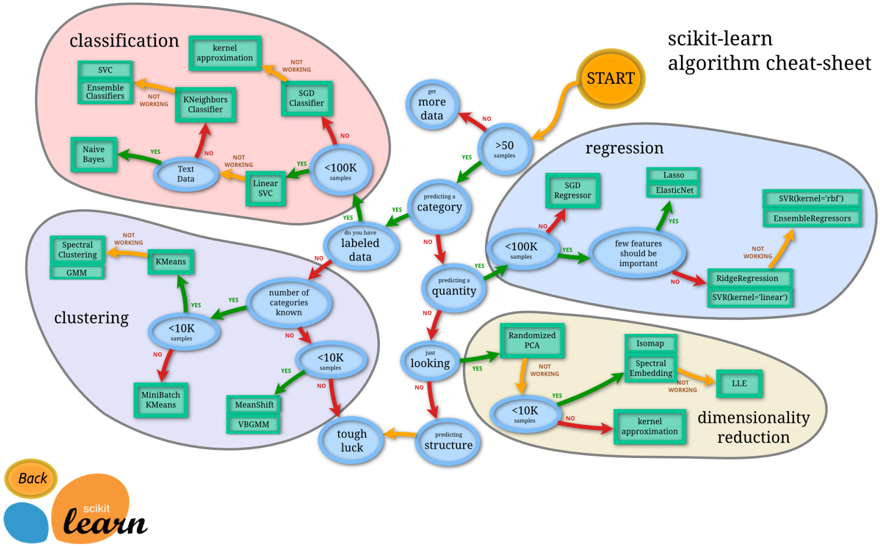
Introduction

- So far, we have studied **batch** kernel methods
- They need to store and compute an $n \times n$ kernel matrix $\mathbf{K}$

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \ddots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \cdots & k(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix}$$

  - SVM, SVR: QP problem
  - KRR, GP: inversion of $\mathbf{K}$
  - Other kernel methods for clustering or dimensionality reduction compute the eigenvectors/eigenvalues of $\mathbf{K}$

High memory and computational requirements


scikit-learn
algorithm cheat-sheet
START
>50 samples
get more data
predicting a category
do you have labeled data
predicting a quantity
just looking
predicting structure
tough luck
classification
kernel approximation
SVC
Ensemble Classifiers
KNeighbors Classifier
SGD Classifier
Naive Bayes
Text Data
Linear SVC
<100K samples
clustering
Spectral Clustering
GMM
KMeans
MiniBatch KMeans
MeanShift
VBGMM
number of categories known
<10K samples
regression
SGD Regressor
Lasso
ElasticNet
SVR(kernel='rbf')
EnsembleRegressors
few features should be important
RidgeRegression
SVR(kernel='linear')
dimensionality reduction
Randomized PCA
Isomap
Spectral Embedding
LLE
kernel approximation
YES
NO
NOT WORKING
Back
scikit learn

- Problems with more than 100K patterns call for optimized implementations of kernel-based algorithms (e.g., LIBLINEAR for classification with linear SVMs)
- The number of support vectors grows linearly with the number of training patterns $\Rightarrow$ Complexity of the trained machine

$$f(\mathbf{x}) = \sum_{i \in SVs} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$

- How to make large-scale kernel machines?
  - Parallel computing: Multi-core, GPU
  - Approximate solutions:
    1. Random Fourier features
    2. Subsampling/sketching/chunking
    3. Low-rank approximations for **K**, e.g., Nyström method

Linear vs. non-linear SVM

- For a linear SVM the optimal separating hyperplane can be expressed in closed form

$$\mathbf{w} = \sum_{i \in SVs} \alpha_i y_i \mathbf{x}_i \Rightarrow f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$

- For a non-linear SVM, the optimal hyperplane in the feature space in general cannot be expressed in closed form
- We can only compute the output score using the kernel expansion

$$f(\mathbf{x}) = \sum_{i \in SVs} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x})$$

- Training and testing are much faster with the linear SVM

Reprinted from: Yuan, G.-X., Ho, C.-H., Lin, C.-J. “Recent Advances of Large-Scale Linear Classification"(2012)

	Linear	**Non-linear** (Gaussian kernel)
Data set	Time / Accuracy	Time / Accuracy
MNIST38	0,1 / 96,82	38,1 / 99,70
ijcnn1	1,6 / 91,81	26,8 / 98,69
covtype	1,4 / 76,37	46.695,8 / 96,11
news20	1,1 / 96,95	383,2 / 96,90
real-sim	0,3 / 97,44	938,3 / 97,82
yahoo-japan	3,1 / 92,63	20.955,2 / 93,31
webspam	25,7 / 93,35	15.681,8 / 99,26

covtype: 581.012 patrones, 54 features
yahoo-japan: 176.103 patrones, 832.026 features

Random Fourier Features

- Standard kernel approach

$$\mathbf{x} \in \mathcal{R}^d \longrightarrow \Phi(\mathbf{x}) \qquad \Phi(\mathbf{x})^T \Phi(\mathbf{y}) = k(\mathbf{x}, \mathbf{y})$$

- We could map the data explicitly to a low-dimensional feature space so that the inner product approximates the kernel function

$$\mathbf{x} \in \mathcal{R}^d \longrightarrow g(\mathbf{x}) \in \mathcal{R}^D \qquad g(\mathbf{x})^T g(\mathbf{y}) \approx k(\mathbf{x}, \mathbf{y})$$

- How to choose a good mapping that approximates $k(\mathbf{x}, \mathbf{y})$?

Bochner Theorem

Any continuous shift-invariant kernel $k(\mathbf{x},\mathbf{y}) = k(\mathbf{x}-\mathbf{y})$ is the Fourier transform of a probability density function $p(\omega)$

$$k(\mathbf{x},\mathbf{y}) = \int p(\omega) e^{j\omega^T(\mathbf{x}-\mathbf{y})} d\omega = \mathrm{E}\left[e^{j\omega^T(\mathbf{x}-\mathbf{y})}\right]$$

- The Fourier transform of a Gaussian kernel is also a Gaussian function
- Since both the probability distribution and the kernel are real, the integral converges when the complex exponentials are replaced with cosines

- Draw D i.i.d. samples $\omega_1, \ldots, \omega_D$ from $p(\omega)$ (a d-variate Gaussian pdf). Note that ω_i has the same dimension as $\mathbf{x}$.
- Draw D i.i.d. samples $b_1 \ldots, b_D$ from a uniform distribution on $[0, 2\pi]$

$$g(\mathbf{x}) = \sqrt{\frac{2}{D}} \left[\cos\left(\omega_1^T \mathbf{x} + b_1\right) \quad \ldots \quad \cos\left(\omega_D^T \mathbf{x} + b_D\right)\right]^T$$

- Then, $g(\mathbf{x})^T g(\mathbf{y})$ is an estimator of $k(\mathbf{x}, \mathbf{y})$
- RFF provide an explicit mapping with some advantages
  1. We can work with a linear SVM/SVR
  2. If $D << n$ we can approximate the kernel matrix by the rank-D matrix $\mathbf{GG}^T$, where $\mathbf{G}$ has the D-dimensional RFFs as rows

Subsampling

- Typically the kernel expansion is sparse (only a few $\alpha_i \neq 0$)

$$f(\mathbf{x}) = \sum_{i \in SVs} \alpha_i k(\mathbf{x}_i, \mathbf{x})$$

- Choose a good subset of training patterns: by KNN or heuristic techniques
- Incremental training, mini-batches, chunking or sketching
  - Split the training data set into N mini-batches: $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_N\}$
  - SVM trained with the first mini-batch $\{\mathcal{S}_1\} \rightarrow \sharp\, SV_1$
  - SVM trained with $\{\mathcal{S}_2, SV_1\} \rightarrow \sharp\, SV_2$
  - SVM trained with $\{\mathcal{S}_3, SV_2\} \rightarrow \sharp\, SV_3$
  - ...

Low-rank approximation of $\mathbf{K}$

- To improve scalabity of kernel methods we can also approximate the rank-n kernel matrix $\mathbf{K}$ by a low-rank approximation $\tilde{\mathbf{K}}$ with rank $r << n$
- Recall that RFFs provide a low-rank approximation $\tilde{\mathbf{K}} = \mathbf{G}\mathbf{G}^T$
- What is the advantage of working with $\tilde{\mathbf{K}}$ instead of $\mathbf{K}$?
  - Memory requirements reduce from $n(n+1)/2$ to nr
  - For KRR we need to compute

$$\boldsymbol{\alpha} = \left(\tilde{\mathbf{K}} + \lambda \mathbf{I}\right)^{-1} \mathbf{y}$$

    if $\tilde{\mathbf{K}}$ has rank $r << n$, its inverse can be computed more efficiently

Nyström method

- Pick $r << n$ columns at random from $\mathbf{K}$

$$\boldsymbol{K} = \begin{array}{cc} & \begin{array}{cc} \boldsymbol{C} & \phantom{\boldsymbol{K}_{21}^T} \end{array} \\ n \updownarrow & \left[\begin{array}{cc} \boxed{\begin{array}{c} \boldsymbol{K}_{11} \\ \boldsymbol{K}_{12} \end{array}} & \begin{array}{c} \boldsymbol{K}_{21}^T \\ \boldsymbol{K}_{22} \end{array} \end{array}\right] \\ & \begin{array}{cc} \overset{\longleftrightarrow}{r} & \phantom{\boldsymbol{K}_{21}^T} \end{array} \end{array}$$

- The low-rank Nyström approximation is

$$\tilde{\mathbf{K}} = \mathbf{C}\mathbf{K}_{11}^{-1}\mathbf{C}^T$$

Online learning

- We have studied **batch** kernel methods
  - Model parameters are obtained from a training data set and kept fixed during testing
    1. **Expansion coefficients**

       $$\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_M)^T$$

    2. **Dictionary** = Support Vectors

       $$\mathcal{D} = \{\mathbf{x}_i, i = 1, \ldots, M\}$$

  - For a new test pattern $\mathbf{x}_n$ the kernel method outputs

    $$f(\mathbf{x}_n) = \sum_{i=1}^{M} \alpha_i k(\mathbf{x}_i, \mathbf{x}_n) = \mathbf{k}_n^T \boldsymbol{\alpha}$$

- In many learning problems patterns arrive sequentially: **online learning**, **sample-by-sample learning**, **sequential learning**

- Both the expansion coefficients and the dictionary change with each new incoming pattern
  1. Coefficients

$$\boldsymbol{\alpha}(n) = (\alpha_1(n), \dots, \alpha_{M_n}(n))^T$$

  2. Dictionary

$$\mathcal{D}_n = \{\mathbf{x}_i(n), i = 1, \dots, M_n\}$$

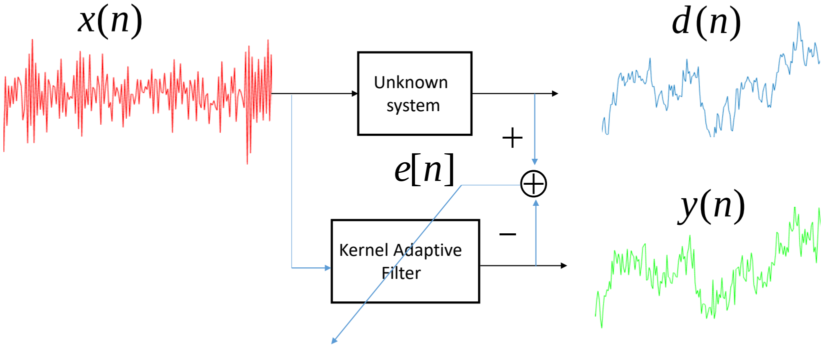
     Note: the dictionary size M_n may change as well!
- The output is

$$f(\mathbf{x}_{n+1}) = \sum_{i=1}^{M_n} \alpha_i(n) k(\mathbf{x}_i(n), \mathbf{x}_{n+1})$$

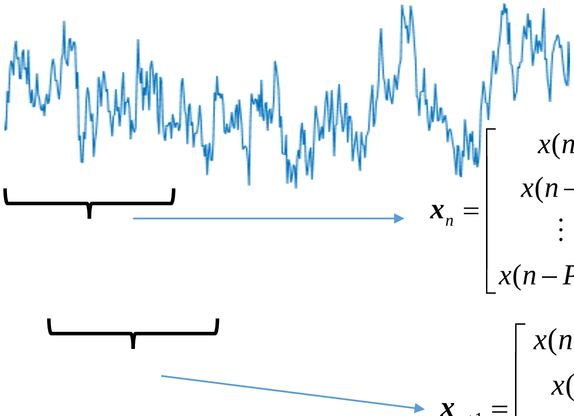
- **Kernel Adaptive Filterig** (KAF) / **Online kernel methods**: Updating algorithms for

$$\alpha_i(n) \rightarrow \alpha_i(n+1) \qquad \mathcal{D}_n \rightarrow \mathcal{D}_{n+1}$$

Kernel Adaptive Filtering (KAF)



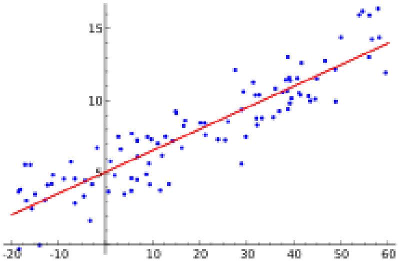
Time-embedding

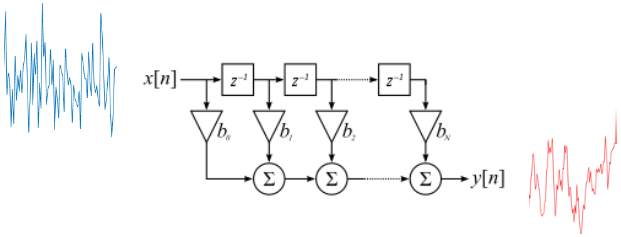


$$\boldsymbol{x}_n = \begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-P+1) \end{bmatrix}$$

$$\boldsymbol{x}_{n+1} = \begin{bmatrix} x(n+1) \\ x(n) \\ \vdots \\ x(n-P) \end{bmatrix}$$

Essentially, filtering is regression with time-embedding





Kernel Adaptive Filtering (KAF) problem

- With data observed up to $n-1$ we build a kernel-based (GP, KRR, SVR) regressor or predictor
- Predictor parameters at $n-1$: $\mathcal{D}_{n-1} = \{\mathbf{x}_i(n-1)\}$, and $\alpha_i(n-1)$
- We observe $\mathbf{x}_n$ and make a new prediction

$$f(\mathbf{x}_n) = y(n) = \sum_i \alpha_i(n-1)k(\mathbf{x}_i(n-1), \mathbf{x}_n)$$

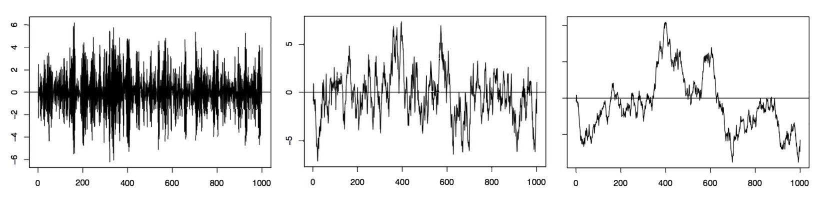
- Then, we get the desired output, $d(n)$, and compute the prediction error: $e(n) = d(n) - y(n)$
- Adaptive algorithm:

$$\alpha_i(n-1) \rightarrow \alpha_i(n) \qquad \mathbf{x}_i(n-1) \rightarrow \mathbf{x}_i(n)$$

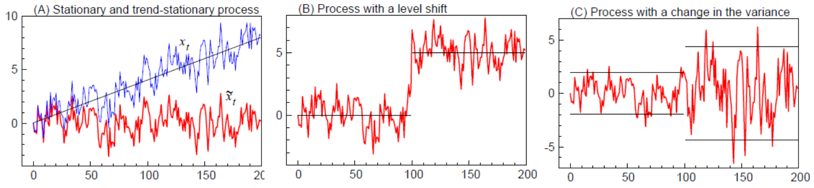
to minimize an error loss function (e.g., $\sum_n e(n)^2$)

Static vs. dynamic/adaptive models

Stationary processes → static models



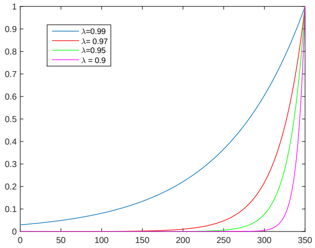
Non-stationary processes → Dynamic/adaptive models



- With non-stationary time series or dynamic models, adaptive algorithms must be able to "forget" the past
- Ability to perform **tracking** under dynamic systems or to react to changes in the input signal properties
- Idea: past data are weighted less than more recent data

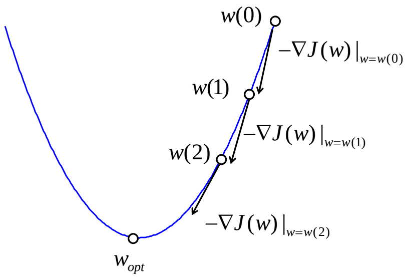
$$Cost = \sum_{k=0}^{n} \lambda^{n-k} e^2(k)$$

where $0 < \lambda < 1$ is an exponential forgetting factor



Least Mean Square (LMS) algorithm

- The most popular adaptive algorithm for linear systems is the **Least Mean Square (LMS)** algorithm (Widrow & Hopf, 1960)
- It is a **Stochastic Gradient Descent (SGD)** algorithm



Formulation

- At time n the vector of regressor coefficients (filter) is

$$\mathbf{w}(n) = \begin{bmatrix} w_1(n) & w_2(n) & \dots & w_P(n) \end{bmatrix}^T$$

- The output is

$$y(n) = \mathbf{w}(n)^T \mathbf{x}_n = \begin{bmatrix} w_1(n) & w_2(n) & \dots & w_P(n) \end{bmatrix} \begin{bmatrix} x(n) \\ x(n-1) \\ \dots \\ x(n-P+1) \end{bmatrix}$$

- In the linear case P is the dimension of the regressor and the time-embedding applied to the input
- We want to minimize

$$J(\mathbf{w}(n)) = \frac{1}{2} e(n)^2 = \frac{1}{2} \left(d(n) - \mathbf{w}(n)^T \mathbf{x}_n \right)^2$$

- The gradient is

$$\nabla J(\mathbf{w}(n)) = \begin{bmatrix} \frac{\partial J(\mathbf{w}(n))}{\partial w_1(n)} \\ \vdots \\ \frac{\partial J(\mathbf{w}(n))}{\partial w_P(n)} \end{bmatrix} = -2e(n) \begin{bmatrix} x(n) \\ \vdots \\ x(n-P+1) \end{bmatrix} = -e(n)\mathbf{x}_n$$

i.e., $\nabla J(\mathbf{w})$ = - error $\times$ input vector

- We update the coefficients as

$$-\mu \nabla J(\mathbf{w}) = \mu e(n)\mathbf{x}_n$$

where μ is the
  - Step-size (signal processing)
  - Learning rate (machine learning)

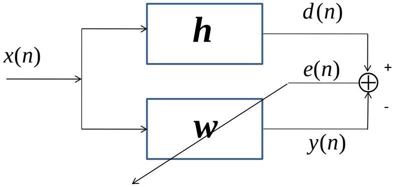
Least Mean Square (LMS) Algorithm

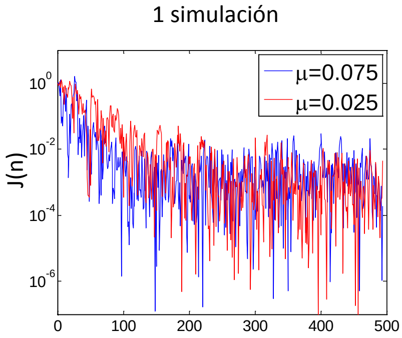
```
Initialize $\mathbf{w}(0) = [0, \ldots, 0]^T$
For $n = 0, \ldots$
    $e(n) = d(n) - \mathbf{w}(n)^T \mathbf{x}_n$
    $\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n) \mathbf{x}_n$
end
```

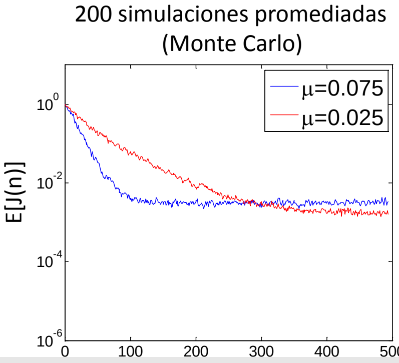
- Advantages:
  - It works with non-stationary signals (tracking)
  - Easy implementation and very low computational cost: $P+1$ multiplications and $P-1$ additions per iteration
- Drawbacks:
  - Slow convergence

Example

Identification of an unknown linear system







Kernel LMS

- Can we apply the LMS algorithm in the feature space?
- The answer is yes and the resulting algorithm is the **Kernel Least Mean Square (KLMS)**
- We apply a linear adaptive filter in the transformed (feature) space
- In the input space we have a non-linear adaptive filter

- The input patterns are mapped to a high-dimensionality space
$$\mathbf{x}_n \rightarrow \Phi(\mathbf{x}_n)$$
- Let us denote the linear regressor or filter in the feature space as $\omega(n)$
- The output (prediction) is
$$y(n) = \omega(n)^T \Phi(\mathbf{x}_n)$$
- After observing the desired output, $d(n)$, we compute the error, $e(n) = d(n) - \omega(n)^T \Phi(\mathbf{x}_n)$, and update the filter coefficients with the LMS
$$\omega(n+1) = \omega(n) + \mu e(n) \Phi(\mathbf{x}_n)$$

- Initialize $\omega(0) = \mathbf{0}$
- Applying the LMS sequentially we get

$$\begin{aligned}
\omega(n) &= \omega(n-1) + \mu e(n-1)\Phi(\mathbf{x}_{n-1}) \\
&= [\omega(n-2) + \mu e(n-2)\Phi(\mathbf{x}_{n-2})] + \mu e(n-1)\Phi(\mathbf{x}_{n-1}) \\
&= \omega(n-2) + \mu\left[e(n-2)\Phi(\mathbf{x}_{n-2}) + e(n-1)\Phi(\mathbf{x}_{n-1})\right] \\
&= \omega(0) + \mu \sum_{k=0}^{n-1} e(k)\Phi(\mathbf{x}_k) = \mu \sum_{k=0}^{n-1} e(k)\Phi(\mathbf{x}_k)
\end{aligned}$$

- The output can be expressed through the **kernel trick**

$$\begin{aligned}
y(n) &= \omega(n)^T\Phi(\mathbf{x}_n) = \sum_{k=0}^{n-1} \mu e(k)\Phi(\mathbf{x}_k)^T\Phi(\mathbf{x}_n) = \\
&= \sum_{k=0}^{n-1} \underbrace{\mu e(k)}_{\alpha(k)} k(\mathbf{x}_k, \mathbf{x}_n)
\end{aligned}$$

- Each new input pattern is added to the dictionary

$$\mathcal{D}_n = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$$

- The coefficient associated to each pattern is $\alpha(n) = \mu e(n)$
- The output is the kernel expansion

$$y(n+1) = \sum_{k=1}^{n} \alpha(k) k(\mathbf{x}_k, \mathbf{x}_{n+1})$$

- **Problem**: The dictionary size and the complexity of the expansion grow unbounded with *n*

Limiting the dictionary growth

- In practice, KLMS algorithms apply different methods to limit the dictionary size
  - Sliding window $\longrightarrow$ fixed budget
  - A new pattern is added to the dictionary only when some criterion is met
  - A popular criterion is the **coherence**

Coherence

Given a dictionary $\mathcal{D}_M = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\}$, and a kernel normalized to fullfil $k(\mathbf{x}, \mathbf{x}) = 1$, coherence is defined as

$$\eta = \max_{i \neq j} |k(\mathbf{x}_i, \mathbf{x}_j)|, \qquad \forall (\mathbf{x}_i, \mathbf{x}_j) \in \mathcal{D}_M$$

Coherence-based KLMS

- Given a size M dictionary: $\mathcal{D}_M$
- For each new input pattern $\mathbf{x}_n$
  1. If $\max_{i \in \mathcal{D}_M} |k(\mathbf{x}_i, \mathbf{x}_n)| \geq \eta_0$
     - Update expansion coefficients

$$\boldsymbol{\alpha}(n+1) = \boldsymbol{\alpha}(n) + \mu e(n)\mathbf{k}(\mathcal{D}_M, \mathbf{x}_n)$$

  2. If $\max_{i \in \mathcal{D}_M} |k(\mathbf{x}_i, \mathbf{x}_n)| < \eta_0$
     - Add $\mathbf{x}_n$ to the dictionary

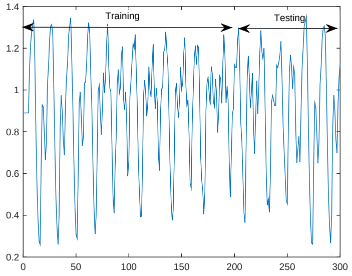
$$\mathcal{D}_{M+1} = \{\mathcal{D}_M, \mathbf{x}_n\}$$

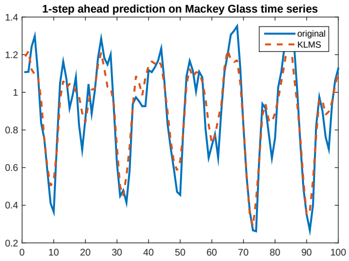
     - Update expansion coefficients

$$\boldsymbol{\alpha}(n+1) = \begin{bmatrix} \boldsymbol{\alpha}(n) \\ 0 \end{bmatrix} + \mu e(n)\mathbf{k}(\mathcal{D}_{M+1}, \mathbf{x}_n)$$

Example

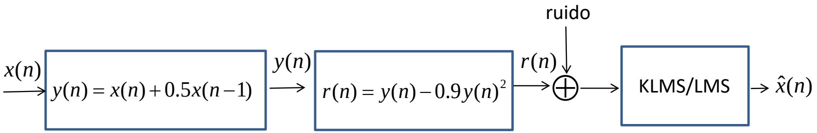
Prediction of the Mackey-Glass chaotic time-series

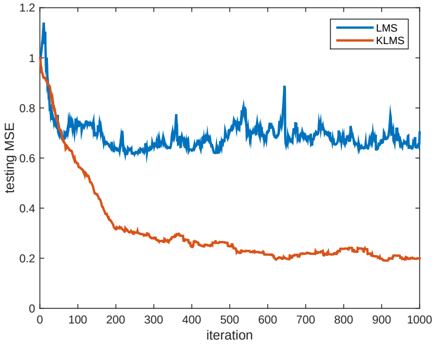




KLMS vs LMS

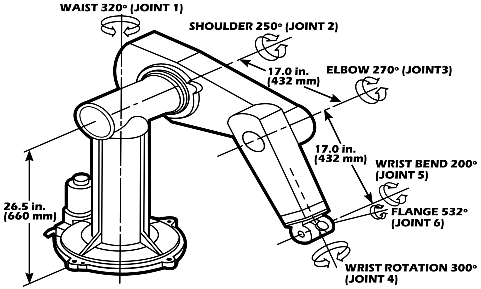
Inversion of a non-linear system (non-linear equalization)





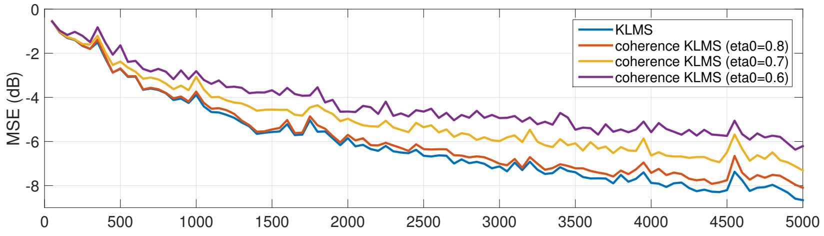
KIN40K dataset

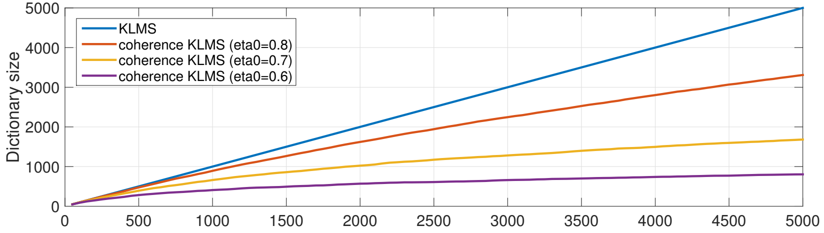
- Forward kinematics of an eight-link all-revolute robot arm



- Data set: 40000 data
  - Input: 8 features (angles)
  - Output: one-dimensional
- Non-linear regression problem

Comparison KLMS vs. coherence-based KLMS





Conclusions

- We can improve kernel methods scalability by
  - Random Fourier features
  - Subsampling/sketching
  - Low-rank approximations of the kernel matrix
- Online kernel methods
  - Sample-by-sample (sequential) adaptation of the model parameters: dictionary+coefficients
  - A popular algorithms is the KLMS: dictionary growth
  - KLMS + coherence criterion: limits the dictionary size
  - Many other KAF algorithms: https://github.com/steven2358/kafbox (toolbox en Matlab)